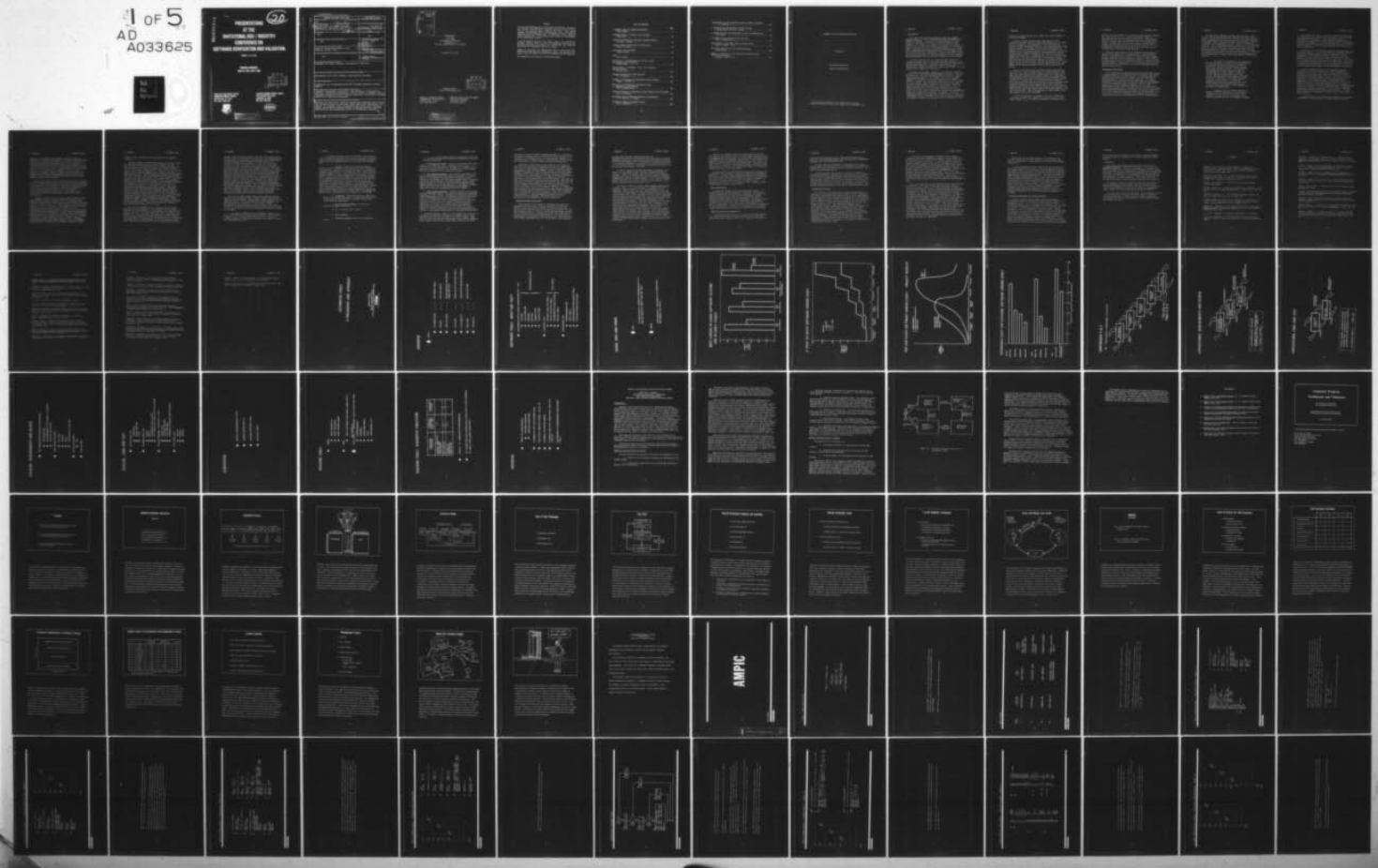
AD-A033 625 ROME AIR DEVELOPMENT CENTER GRIFFISS AFB N Y F/G 9/2
PRESENTATIONS AT THE RADC/AR³A INVITATIONAL DOD/INDUSTRY CONFER--ETC(U)
AUG 76

UNCLASSIFIED NL

1 OF 5
AD
A033625



ADA033625

20

PRESENTATIONS AT THE INVITATIONAL DOD / INDUSTRY CONFERENCE ON SOFTWARE VERIFICATION AND VALIDATION

AUGUST 3, 4, 5, 1976

PROGRAM MANAGER
JOHN M. IVES, CAPT, USAF

DDC
RECEIVED
DEC 21 1976
D

ROME AIR DEVELOPMENT CENTER
INFORMATION SCIENCES DIVISION
GRIFFISS AFB, NY 13441
POC: CAPT JOHN IVES

ADVANCED RESEARCH PROJECTS AGENCY
1400 WILSON BOULEVARD
ARLINGTON, VA 22209
POC: BILL CARLSON

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: N/A	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): PRESENTATIONS AT THE INVITATIONAL DOD/INDUSTRY CONFERENCE ON SOFTWARE VERIFICATION AND VALIDATION	5. TYPE OF REPORT & PERIOD COVERED: (3-5 Aug 76) Conference Proceedings
	6. PERFORMING ORG. REPORT NUMBER: N/A
7. AUTHOR(s): N/A	8. CONTRACT OR GRANT NUMBER(s): N/A
9. PERFORMING ORGANIZATION NAME AND ADDRESS: N/A	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 62702F 5581 02 73
11. CONTROLLING OFFICE NAME AND ADDRESS: Rome Air Development Center (ISIS) Griffiss AFB NY 13441	12. REPORT DATE: 3-5 Aug 76
	13. NUMBER OF PAGES: 425
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office): N/A	15. SECURITY CLASS. (of this report): Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)
Recommend for Public Release, Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)
Recommended for Public Release, Distribution Unlimited

18. SUPPLEMENTARY NOTES
Conference was co-sponsored with the Advanced Research Projects Agency

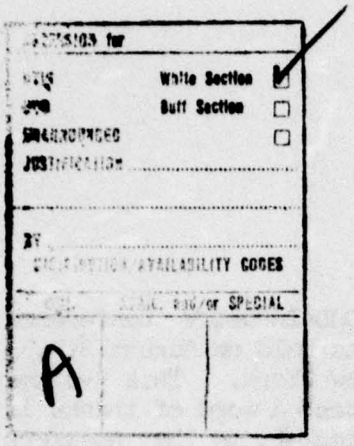
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
Computer Programming, Validation, Certification, Verification, Software, Structured Programming, Quality Control, FORTRAN, JOVIAL, Computer Software Security, Symbolic Execution, Self-Metric Software Assertions Programming Languages

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)
Original and previously published papers are presented that define the state-of-the-art in software verification and validation. The papers, in general, cover the spectrum from large scale software system verification through working validation tools to Symbolic Execution and the theoretical proof of correctness advances.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

309050

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)



Presentations at the RADC / ARPA Invitational DOD/Industry Conference on Software Verification and Validation

August 3,4,5, 1976

DDC
RECEIVED
DEC 21 1976
D

PROGRAM MANAGER
John M. Ives, Capt, USAF

Rome Air Development Center
Information Sciences Division
Griffiss AFB, NY 13441
POC: Capt John Ives

Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, VA 22209
POC: Bill Carlson

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

PREFACE

The first RADC/ARPA-sponsored Invitational DOD/Industry Conference on Software Verification and Validation was held on August 3,4,5, 1976 at the Syracuse Hotel, Syracuse New York. This volume contains presentations made at the conference. A word of thanks is expressed to all paricipants, and , in particular, to the speakers whose presentations are contained in this volume for contributing to the conference's success.

A special thanks is given to Col Robert Krutz, the Information Sciences Division Chief of Rome Air Development Center for his Welcoming speech which indicated the philosophies and predicted directions the Information Sciences are headed.

Thanks is also due to Ms. Deborah Danti and Mr. Stanley Borak for successfully operating the registration desk, and working long hours behind the scenes that kept the conference running smoothly.

This document is not printed at government expense.

TABLE OF CONTENTS

	Page
A Planner's View of Program Verification Carl Engelman	1
Automated Tools: A Philosophy and Approach Barry Press	24
Needs of Verification in Ballistic Misssile Defense Maj Ben A. Johnson	41
Computer Program Verification and Validation Donald J. Reifer	48
Consistency Checker, Summary Michael Landes	69
AMPIC Mike Ikezawa	71
Verification of Requirements and Design through Structured Analysis Techniques Raymond Rubey	96
New Directions in Automated Tools for Improving Software Quality Leon G. Stucki	101
Software Testing Tools (JAVS and RXVP) W. R. Wisehart	143
Advances in Technology for Verification and Validation Edward F. Miller, Jr.	167
The Impact of Verification and Security Goals on Programming Language Design Ann B. Marmor-Squires	199
Symbolic Execution: A Compromise between Testing and Verification John A. Darringer	213
Self-Metric and Self-Checking Software and Validation S. S. Yau and R. C. Cheung	231
Integrated Software Development System Roy W. Coppinger	263

Verification and Error Detection Using a Symbolic Evaluator
T. E. Cheatham, Jr. 287

A System for the Verification of JOCIT Programs
Bernard Elspas and Jay M. Spitzen 294

A Program Verifier with Assertions in terms of Abstract Data
Val Schorre 314

An Example of Hierarchical Design and Proof
Jay M. Spitzen, et al. 344

Verification of Data Types Using Algebraic Axioms
David R. Musser, et al. 370

An On-Line Session with the Stanford Verifier
David Luckham 392

Gypsy: A Language for Specification and Implementation of Verifiable Programs
Donald I. Good, et al. 407

A PLANNER'S VIEW OF PROGRAM VERIFICATION

C. Engelman

The MITRE Corporation

Bedford, Massachusetts

This work was sponsored by the United States Air Force
Electronics System Division under Contract No. F19628-76-C-0001.

INTRODUCTION

This paper was originally prepared as a portion of MITRE Technical Report, MTR-3232, SOFTWARE DEVELOPMENT REQUIREMENTS FOR COMMAND AND CONTROL, in support of the Technological Planning activity in the Electronic Systems Division Development Plans Deputate. That report, not released in its entirety for public distribution, is intended to provide the background and rationale to support the definition of the ESD Technology Needs in the software area. These Technical Needs in turn will provide guidance to the Air Force Laboratories, especially the Rome Air Development Center, as to needed technology developments to rectify deficiencies and/or provide new capabilities for the software aspects of the automatic data processing function in Air Force Command and Contol Systems.

The words "Verification and Validation" have become both cliches and budget items in large system software programs. These informal approaches have in particular been developed vigorously by the aerospace industry (cf. Reifer74), where the practical necessities of missile software do not permit postponement until the advent of more formal proof techniques.

We shall in this paper, however, be concerned only with the possibility of demonstrating mathematically, at least at some level of abstraction, that software enjoys certain properties. Among these properties might be, for example, the fact that routines, regarded as functions, are well defined over certain argument domains or, regarded as procedures, terminate for a certain class of initial values. Most theoretical work, starting with Floyd67, has been directed towards proving the correctness of programs, defined in terms of user-supplied I/O relationships. Here, the least mechanical part of the procedure turns out to be the generation of the inductive assertions associated with program loops. Even there, work is progressing towards providing mechanical assistance in assertion generation, cf. German75. The most impressive system for the completely automatic proof of properties of programs is that of Moore73. All current successes in the proof of program correctness are restricted, from the viewpoint of the

software associated with Air Force Command and Control systems, to quite small programs.

The most conservative approaches today seem to be the restriction of proofs either to quite simple properties or to small, critical subsystems. The current work on the development of verifiably secure operating systems for multi-security-level on-line computer systems, cf. Burke74, utilizes both these approaches. The property chosen for verification, the so-called *-property, cf. LaPadula73, is a quite simple property that constitutes a sufficient condition that the system conforms to the Department of Defense security policy, Rush72. Furthermore, in order to facilitate the design, the proofs, and the implementation, all data references are monitored by a "security kernel" that, as an abstract machine, is the sole source of such services available to the more external balance of the operating system. By this localization of all "real" memory references, it is hoped to greatly reduce the size of that portion of the operating system for which the *-property must be proved.

Work in this direction is obviously of great significance to the Air Force. Tactical Command and Control systems must provide users with restricted access privileges to data on terminals that might even fall (together with informed users) into enemy hands. Multi-security-level on-line systems are also being sought for the economic benefits they provide through continuous resource sharing in contrast to conventional, costly "sanitization" procedures. Multi-level secure on-line data processing or communications systems currently operating or under construction include the MULTICS Systems at the Air Force Data Services Center, the Security Kernel Based MULTICS System, the Prototype Secure Front End Processor, the PDP-11/45 Security Kernel Based System at The MITRE Corporation, and the AUTODIN system. Those under development include SATIN IV and some upgraded versions of the AABNCP.

It would be desirable, if feasible, to be in a position to prove other, perhaps more complicated, properties of such multi-level systems. It is, in reality, necessary to demonstrate more

generally that the integrity of a data processing system is assured, e.g., that no user can, in an unauthorized manner, deny the legitimate services of the system to other users. Software which supports high cost/high risk operations such as space missions or satellite launches must work the first time. For communications systems, we might want to prove that, at least, the President's message will get delivered and to the correct recipient. It would be preferable, though more difficult, to prove that it will be delivered within a stipulated time interval.

There are certain areas in which software probably cannot be used unless we actually go further and prove it more generally correct. One example would be software that provides an automatic strategic weapon allocation capability. Another would be software entrusted with the actual encryption of messages. Until software is demonstrated correct, it probably should not be employed for such critical purposes. Furthermore, it is probably only such critical services that could justify the cost of full correctness proofs in the foreseeable future.

Mechanical Verification

Since combinatorial explosion precludes the exhaustive testing of large programs, it is tautological that testing can only reveal the presence of bugs, not their absence. The pressing need felt today for software verification derives, not from such logical considerations, but rather from bitter experience. All large programs have had their share of bugs, and legion are the anecdotes of software systems that have functioned for several years before some novel utilization revealed fundamental errors. More significantly, "tiger teams" have been able to crack the security and to gain complete control, including access to all files, of such current operating systems as OS (Goheen72) and MULTICS (Karger74). It is these frightening experiences that have, in particular, led to the search for methods of designing and implementing verifiable security kernels for on-line multi-security-level computer systems.

The same sort of empirical evidence that dictates a lack of faith in non-verified software also dictates that, with respect to matters involving national security, we have no business trusting manual mathematical proofs as sufficient verification of properties of programs. There is an equally large treasury of anecdotes relating to false mathematical proofs. This is a history, we might point out, that, unlike the history of software errors, extends over centuries. We are referring to submitted, refereed, published results accepted for a number of years before either they were discovered false or a gap in the proof was disclosed. To choose but one example, we should like to quote from the introduction of an article appearing in a recent issue of the Annals of Mathematics, Masur75:

> The study of the geometry of the classical Teichmueller spaces was begun in 1959 by Kravetz [9]. The starting point was the classical theorem of Teichmueller on extremal quasiconformal maps between compact Riemann surfaces. The Teichmueller theorem was used to argue that with respect to the Teichmueller metric, Teichmueller space is straight and that it has negative curvature. In turn, negative curvature was used to show that every finite subgroup of the Teichmueller modular group has a fixed point. This latter statement is known to be equivalent to the Nielsen Realization Problem which conjectures that every finite subgroup of the mapping class group of a surface can be realized by a finite subgroup of the group of homeomorphisms.
>
> Recently Linch [10] found a mistake in Kravetz's curvature arguments so that problem and consequently also the fixed point problem were reopened. The main result in this paper is that Teichmueller space does not have negative curvature.

The history here, then, is that the published proof survived for twelve years until it was discovered faulty. It took another four years before the result itself was discovered false. Unfortunately, with respect to the validity of demonstrations of properties of programs, the history might be reversed. A manually verified security kernel may first be demonstrated insecure--by an enemy's cracking it--and only subsequently may the error in the proof be discovered.

A discussion of the precarious certainty of mathematical proof is found in Davis72. It is replete with both histories of major errors by first rank mathematicians and mathematical texts accompanied by tens of pages of errata. We must further keep constantly in mind that ordinary published mathematical proofs probably normally range from ten lines to twenty pages. The proof of so simple a property as the security preservation of the kernel of a practical operating system is, on the other hand, likely to be a document some inches thick. Such massiveness will do little to instill confidence with respect to the absoluteness of any proof .

We are convinced that there is an imperative that the verification of critical properties of critical programs be pronounced by machines. While there are many difficulties to overcome before this becomes a practical alternative, a matter discussed at some length below, we should like to point out that there exists at least one domain in which the superiority of mechanical over the manual verification has become almost commonplace. Every entry in symbolic mathematical tables, e.g., tables of definite or indefinite integrals, as well as every symbolic computational exercise in every physics or engineering text represents a theorem. Since 1964 mechanical symbolic computation systems such as MATHLAB, Engelman71, or SIN, Moses71, have been discovering errors in well-renowned tables and texts. In some cases, these have corresponded to false proofs; in other cases, to false theorems.

The most desirable of situations with respect to the automatic verification of the correctness of programs would be for the

programs to be automatically generated by theorem provers as exemplified by Green69 or Waldinger69. The work of Engelman74 would represent a middle position in which the validity of each of the pattern based program transformation rules could be proved independently and the validity of any generated program would then follow as a consequence. Short of such automatic program synthesis schemes, the ideal in mechanical program verification would be for the proofs to be generated by a mechanical theorem prover, cf. Moore73. Unfortunately, there is little evidence that any such schemes show promise of developing within the next decade to the stage that they can handle anything approaching current systems software or applications programs.

While we must remain skeptical about the capability of theorem provers to generate demonstrations of properties of large programs--in fact there are quite discouraging complexity results with respect to theorem provers, cf. Fischer74--it may remain possible that much can be done in the way of providing mechanical verification of proofs, constructed either manually or in semi-automatic environments, employing mechanical aids. For example, proof checkers for the first order predicate calculus are available (Weyrauch74).

Of great interest are several systems under development that are attempting to find a middle ground between automatic proof generation, which seems too remote a goal, and mechanical proof checking, which places too tedious a burden on the person supplying the proof. These systems, all of which appear quite similar on the surface, are efforts to create a program verification environment in which the user has on-line access to a variety of tools: bookkeeping, proof checking, automatic simplifiers and, especially, verification condition generators. The most prominent among those systems that exist in experimental form are those at the IBM Watson Research Center (IBM), King75, the Information Sciences Institute, University of Southern California (ISI), Uncapher75, pp. 1-13, the Stanford Artificial Intelligence Laboratory (SAIL), Luckham76, Stanford Research Institute (SRI), Elspas76, Systems Development Corporation (SDC), Schorre75, the University of Texas at Austin

(TEXAS), Good74, and XEROX Palo Alto Research Center (XEROX), Deutsch73.

We shall attempt a composite description of how these systems are intended to operate. All are based on Floyd's inductive assertion method of proving properties of programs, Floyd67. The first portion of each system is a parser that accepts programs written in a specific language (IBM: a subset of PL/I; ISI and SAIL: PASCAL; SDC: A LISP-like language; SRI: a subset of JOVIAL/J3; TEXAS: GYPSY) as well as certain formal statements about the program supplied by the user. These include input predicates and output predicates that define the "correctness" of the program and a set of inductive assertions associated with the program loops. These are translated into some appropriate internal form. The translated program and annotations are then passed to a verification condition (VC) generator. The output of this generator is a set of verification conditions, i.e., statements in some first order predicate calculus generally involving arithmetic, equalities, inequalities, and some programming structures such as stacks, arrays, or lists. That is, the verification generator maps the problem of proving properties about programs into a problem of proving statements in some formal logic. In order to do this, the system must contain axiomatic knowledge about the semantics of the programming language. This explains the popularity for these systems of PASCAL, a language designed for structured systems software production, amenable to formal axiomatization, cf. HOARE73. The VCs generated depend strongly on the particular inductive assertions originally supplied by the user.

The proof of correctness, i.e., the proof that the program satisfies the output predicate on the condition that the input satisfies the input predicate, is thus mechanically reduced to the proof of the VCs. It is this latter proof that is intended to be interactive. For this purpose, the system must possess first some basis of facts (axioms or theorems) expressed in the VC calculus. These facts may be supplied as part of the system, may be deduced by the system (alone or interactively), or may be introduced by the user. Operating on this basis of facts is a deductive system that

strives to find proofs that the VCs follow from the factual basis. In this process, it may be guided by the user. The mechanical tools provided tend to be a melange of theorem provers and "black boxes". The theorem provers may be complete for some calculi, e.g., resolution principle theorem provers for the pure first order predicate calculus or a semi-decision procedure for Presburger Arithmetic, or may be heuristic goal-oriented processes applied to non-semi-decidable domains such as integer arithmetic. The black boxes may include tautology deciders for the propositional calculus (Weyrauch at SAIL has one that can decide a tautology containing over 100 variables in seconds) and simplification procedures for algebraic and conditional expressions. In some systems, the user is allowed to introduce facts without proof (assumptions) and some post-processing in the form of a proof checker (cf. Weyrauch74) is needed to confirm the verification. Finally, these interactive systems present varying capabilities for automatic bookkeeping to help manage larger proof efforts. This is particularly important in the case of hierarchical programs where the proof is likely to be an extended, incremental process. What is needed in that case, in fact, is an automatic record of how the proof hangs together, so that only the logical minimum need be proved anew when the hierarchical program and/or the proof strategy is modified.

The status of these systems today is that they have been put together on an experimental basis, largely from previous software components and are capable of supporting, if rather ponderously, the demonstrations of the correctness of rather small programs, e.g., sorting algorithms. We should like now to list a number of deficiencies of these systems as we perceive them. This is not intended as condemning criticism, only as indications of directions in which we are sure the authors would agree more development should be encouraged.

1. The size of programs that can be handled. While it is hoped (and occasionally claimed) that the support of proofs layered hierarchically in levels of abstraction will permit the semi-automatic mechanical verification of systems size programs, this is still an unredeemed promise.

2. The type of programs that can be handled. Most programs that have been verified with the use of these systems are basically simple mathemtical algorithms, e.g., sorting algorithms or parsers. Among the systems programming areas in which more work is needed are:

a. Pointers. This area is extremely active. Luckham (SAIL) is working on verification for pointer manipulation in PASCAL, which supplies pointers as data types. The problem is presumably more difficult in a language like PL/I, where a variable may be a disguised pointer. Boyer and Moore (SRI) are working on automatic proofs for pointer manipulating LISP programs. In pure LISP, of course, the pointer manipulation is an invisible property of the implementation that need only be addressed at the level of the verification of the LISP interpreter, not at that of the verification of LISP programs. However, Boyer and Moore wish to extend their automatic proof generator to an extended LISP that admits user invocation of the functions RPLACA and RPLACD. Crocker (ISI) is developing formalisms for the manual verification of pointer manipulating routines in assembly language as part of his general goal of developing proofs of properties of the ARPANET IMP software, e.g., guaranteeing that a message will be delivered.

b. Concurrency. Another active area. TEXAS, for example, has included concurrency and synchronization dictions in the design of GYPSY in order to provide a vehicle for verified systems programming.

c. Actual machine arithmetic in contrast to idealized mathematical domains.

d. I/O. Seemingly always ignored.

e. Error handling.

f. Heuristic Programs for which "correctness" is elusive.

g. At the extreme, McCarthy and Weyrauch (SAIL) wish to develop verification techniques for programs that "interact with the real world".

3. Properties of programs. Proofs of correctness often ignore questions of termination. We also need approaches to the verification of the performance of programs, e.g., guarantees that concurrent processes will not only not deadlock, but that certain ones will be completed within a specified interval.

4. More integrated systems. The need for integration of the verification process is the main theme of this paper. It applies equally to the semiautomatic verification systems we are now discussing. For example, verified PASCAL programs should be compiled by verified PASCAL compilers. More particularly, though, certain questions arise out of the way in which these systems have been slapped together from existing components. For example, one might prefer the construction of axiomatically defined algebraic simplifiers to the use of ad hoc systems such as REDUCE.

5. Better human interfaces. This may turn out to be the make or break issue with respect to the eventual frequent and successful employment of these systems. Included is the need for conceptually simple bookkeeping tools for the management of hierarchical proofs as well as the deeper question of user comprehension of the verification conditions. While the representation of first order predicate calculus formulas in clause form--with existential quantifiers replaced by Skolem functions and both universal quantifiers and conjunctions implicit--may be necessary for a resolution principle black box within the system (cf. Nilsson71, Chapter 6), it is not necessarily the form most comprehensible by the user. Some systems either maintain or resupply the quantifiers.

The deeper question, however, is the traceability of the VCs to the inductive assertions. We expect, for example, that a common occurrence would be for the interactive system to present a user with VCs that it had failed to demonstrate and ask him for guidance. After a short while, the user might realize that this set of VCs

could not, in fact, be proved in general and that it is necessry to go back and strengthen some of the inductive assertions that he had annotated to the program loops. But which and how? Essentially, the system must be somehow capable of "decompiling" the work done by the VC generator in order to supply the required traceability.

Furthermore, much may have to be learned not only about the capabilities we can expect in mechanical verification systems but also about the complementary questions of how to structure, specify, and implement large, complex programs in order to facilitate their mechanical verification. Work at SRI relevant to the structuring of programs as complex as security kernels is reported in Robinson75 and Robinson75a. They employ the technique of Parnas specifications (Parnas72) to define the program as a hierarchy of levels of abstraction, each represented by an abstract machine. The intent is that this structuring and specification technique will abet both the construction of secure systems and the proofs of their security, the proofs themselves being layered by the identical hierarchy. A realistic appraisal of the likely power of mechanical validation tools will probably lead to the concession that the proof techniques must be adapted to the limitations of the semiautomatic verifier and that the program specification technique, in turn, must be adapted to the restrictions of the proof technique. That is, it would not seem reasonable to expect a posteriori mechanical proofs of arbitrarily written programs. The technologies of proof, specification, and implementation must be integrated.

Complete Software Verification

Our discussion of the problem of verifying the kernel of a multi-security-level on-line computer system till now has been in general outline that being followed by the ESD project on the Security Kernel Based Multics System involving ESD, MITRE, MIT, SRI, and Honeywell (Burke74). That is, we have been assuming a security policy enunciated by DoD, a mathematical model that formalizes that policy, a kernel specification that adheres to that model, and an implementation that adheres to that specification. The verification of the kernel would then consist of a proof that the specification

satisfies the model and a second proof that the implementation satisfies the specification--this latter to be more accurately viewed as a hierarchically structured sequence of proofs. To this we have added our own conviction that the verifications need to be checked mechanically. Appropriately, we understand that SRI wishes to commit itself to semiautomatic mechanical verification for the future of that project (Levitt76).

In order to complete the chain of software verification, we should start by asking in which language the implementation is written. It would seem unlikely that this should be the machine language. There is a tendency today to write systems in languages such as BLISS (Wulf70) or PASCAL (Wirth75) that provide structured programming support for systems software. In the case of a MULTICS kernel, PL/I may be difficult to resist since the balance of the MULTICS operating system is written in it.

The use of a higher (or medium) level language presents three problems. First, it is necessary that the language itself have a rigorously defined semantics so that the program proofs are meaningful. Second, there must be a means of transition from the proofs for the implementation language programs to the verification of the machine code. There would seem to be two distinct approaches available to deal with this transition. One would be an additional layer of proofs that the machine code produced by the compiler (or some hand modified version of that code) corresponds to the implementation language code. The other approach would be to make a capital investment in the verification of the correctness of a compiler for the implementation language. This is certainly the more attractive alternative since changes in the system specification and/or the verification effort itself are likely to impose the need for changes in the program. Third, the hardware instructions must themselves bear a formally defined semantics in order that the verification of the programming language to machine language transition be definable. Actually, the need for formal machine language semantics would obviously exist even if the subject program were written in machine language. We shall return to this point below.

Finally, the mechanical verifier, itself, must be verified. This leads to the conclusion that it must be simple enough to permit a convincing manual verification (or a proof within the formal scope of a manually verified mechanical proof-checker). The price of this might be a weakness in the tools provided and would be reflected in limitations on the permissible proof methodology. This latter would serve in turn to restrict the permissible program specification and implementation techniques. The final effect might be to measurably increase the cost of verified software.

A possible solution might derive from our tacit assumption that a program verifier is capable of proving programs more complicated than itself. It might thus be possible to approach a more powerful verifier through a bootstrapping sequence of verifiers of which only the initial one would require a manual verification. However, we must keep in mind that we are dealing here with a rather encompassing definition of correctness, not some relatively simple property such as security preservation.

Kernel Minimization

We know that automatic proof generation grows super-exponentially with program complexity (Fischer74). We have been tacitly assuming that the mechanical (and manual) verification of program properties can be restrained by hierarchical program specification techniques, such as those expounded by SRI, to bearable, though extremely tedious, magnitudes. The success of such an approach to the problem of the verification of the security preservation of complex operating systems might depend on our ability to restructure such systems so as to minimize the complexity of the required kernel. Perhaps a less ad hoc study of operating systems architecture from this point of view would be profitable.

Impact of Distributed Computation

Distributed computation, e.g., to support survivable command and control systems, may introduce additional issues into the general problem of software verification. These are difficult to

anticipate since the logic of such computation systems and the nature of the associated software, even the appropriate programming languages, are little understood today.

It may be that the utilization of integrated circuit based microprocessors will transfer some of the "program" from software to hardware and this might introduce novel verification problems. This point is mentioned again below.

Operating systems often consist, of course, of a number of simultaneous processes. Parallelism introduces special difficulties for verification techniques, and some study has been made of this problem, e.g., Lipton75, Ashcroft73, or Levitt72. Nonetheless, this added complexity might serve to motivate an attempt to restrict the parallelism of the security kernel, as contrasted to that of the extended operating system, of a multi-level computing system. Such options may not be available in the case of distributed processing.

Hardware Verification

The question of the verification of the correctness of a program or of its enjoying any other property, such as termination or security preservation, must resolve finally to the same property being enjoyed by a machine language representation of that program. We must therefore obtain rigorous formalizations of the semantics of machine instruction sets. A partial effort in this direction is, in fact, included in the machine language handbook for at least one computer, the PDP-11/45. That description, based on the specification for each instruction of its effects on a fixed set of state variables, might suffice with perhaps a bit more formal description of the process interpreter, including interrupt handling. On the other hand, it could be argued that a complete proof would also involve the axiomatization of the arithmetic and logical operations. The thought of carrying the proofs of properties of complex programs down to such detail is rather intimidating, though Crocker (ISI) has some such ambitions. Significant work on the verification of microcode is in progress at the IBM Watson Research Center.

In the case of microcomputers in which certain normally software functions are delegated to hardware, we must extend our proof methods to the circuitry just to obtain a proof at the level of current program verification. In the hierarchical approach to program validation being studied at SRI, the suggestion has been made that the "program" be structured and verified without concern as to which levels of abstraction represent software and which hardware, with the boundary being drawn a posteriori. However, most microprocessors will be purchased from manufacturers with no customer control over their internal logic. Even if the logic is formally specified, its implementation might turn out to be dictated by engineering and production considerations. There may be questions therefore as to whether it would necessarily exhibit the "good structure" we assume in software designed with verification in mind.

There is another issue in hardware verification that must also be dealt with. It is the question, especially with respect to multi-level secure systems, of trusting conventional hardware to meet its specifications. Errors in hardware design may invalidate permanently any software verification while hardware failures may do so temporarily. One response to this problem is the so-called "subverter", that is, the purposeful introduction of a process into the operating system whose purpose is to periodically test the correctness of the hardware. This has been done for the testing of the memory access protection mechanisms in MULTICS operating systems. The extension of this technique to properties other than security preservation is far from clear. Even in the case of security preservation, the verification of the kernel depends on the validity of all computer instructions, not just the memory access mechanisms. More safety could perhaps be achieved by the introduction of a "software subverter" that tried periodically to run through many of the kernel routines permuting the values of the significant state variables. More evidence as to what accidents might have occurred could be obtained through the complete audit of all memory accesses, cf. Engelman76.

Perhaps the most promising attack on this problem is the facility incorporated in GYPSY, Ambler76, for specifying dynamic consistency checks and incorporating these into the static program verification.

The final hardware problem, and perhaps the most difficult, is that of the hardware Trojan horse. This is the residual paranoia that remains after all the software has been mechanically verified, the verifier has been verified, the compiler has been verified, the communication lines adequately encrypted, the personnel subject to rigid security clearance procedures, etc. How then can we be sure that the computer manufacturer is not a spy? How can we be sure that some machine instruction, disguised as, say, an integer multiplication, does not--in addition to its purported function--periodically copy a secret file into a non-classified area? Perhaps, the most reasonable answer to this question, if one really wants to deal with it, is that it should be possible to devise acceptance tests that would reduce the probability of such a hardware Trojan horse going undetected to a value lower than our reasonable expectation that cleared personnel are engaging in espionage.

The Cost Effectiveness of Verified Software

Finally, we would like to suggest that as the software verification technology progresses, we try to distill some of our experiences so that, at the start of each Air Force software development project, we could estimate the additional costs involved in verifying the security, the integrity, the correctness, etc. of the system. Cost effectiveness is much more elusive. Having estimated the cost of verification, how do we assign a <u>value</u> to the verification. We have in mind particularly those cases in which the requirement for verification derives from operational necessities. In the case of the automatic allocation of strategic weapons, a decision on which our national survival may depend, what is the value of knowing a program is correct? In the absence of a proof, must we try to function without the automatic capability or is there some level of plausibility at which we employ an unverified program?

The national survival argument as an absolute is quite persuasive, but national security arguments do not produce unbounded defense budgets.

Conclusions

The primary area needing more development is that of an integrated mechanical verification technology. Proofs cannot be mechanically generated. They must be mechanically verified. The proof techniques must be integrated with the mechanical verification technology. The program specification techniques must be integrated with the proof techniques. The implementation technology must be integrated with the specification techniques, the proof techniques and, preferably, a verified compiler. The compiler should be verified with respect to formal machine instruction set semantics.

In addition, we must be mindful that for the verification of complex programs to become an other than rare occurrence, the verification technology must support not only the verification of a fixed, correct program but must equally support the programming effort through its normal debugging and maintenance phases.

The other area of profound importance, in our estimate, is that of the demands placed on the program verification technology by distributed processing. However, we would suggest postponing development here until more is understood of the emerging hardware and software structures.

REFERENCES

Ambler76 Ambler, Allen L., et al, "GYPSY: A Language for Specification and Implementation of Verifiable Programs", Minutes Invitational Software Verification and Validation Conference, Syracuse, N.Y., 3-5 August 1976.

Aschroft73 Ashcroft, E. A., "Proving Assertions about Parallel Programs" Research Rep. CS-73-01 Dept. of Applied Analysis and Computer Sci., University of Waterloo, 1973.

Burke74 Burke, Edmund L., "Synthesis of a Software Security System" PROC. ACM74.

Davis72 Davis, P. J., "Fidelity in Mathematical Discourse: Is One and One Really Two?", Am. Math Monthly 79, 3, (March 1972), pp. 252-263.

Deutsch73 Deutsch, L. P., "An Interactive Program Verifier", Rep. CSL-73-1, XEROX PARC, Palo Alto, CA, May 1973.

Elspas76 Elspas, B.; Boyer, R. S.; Shostak, R. E.; Spitzen, J. M., "A Verification System for JOVIAL/J3 Programs", Technical Report 3756-1, Stanford Research Institute, January 1976.

Engelman71 Engelman, C., "The Legacy of MATHLAB 68", Proc. Second Symposium Symbolic and Algebraic Manipulation, (ACM) Los Angeles, March 1971, pp. 29-41.

_____74 Engelman, C., personal communication. Cf. Grosso74.

_____76 Engelman, C., "Audit and Surveillance of Multi-Level Computer Systems", MTR-3207, MITRE Corporation, Bedford, MA, March 1976.

Fischer74 Fischer M. J.; Rabin, Michael O., "Super-Exponential Complexity of Pressburger Arithmetic" in Karp, Richard M. (Ed.), Complexity of Computation American Mathematical Society, Providence, 1974.

Floyd67 Floyd, R., "Assigning Meanings to Programs" in Schwartz, J. T. (Ed.) Proc. Symp. Applied Mathematics, Vol. 19, American Mathematical Society, Providence, 1967, pp. 19-32.

German75 German, Steven M.; Wegbreit, Ben, "A Synthesizer of Inductive Assertions" IEEE Trans. Software Engineering, March 1975, pp. 68-75.

Goheen72 Goheen, S. M.,; Fiske, R. S., "OS/360 Computer Security Penetration Exercise" WP-4467 The MITRE Corporation, Bedford, MA, October 1972.

Good74 Good, Donald I., "Provable Programs and Processors" Proc. NCC 1974 pp.357-363.

Green69 Green, Cordell, "Application of Theorem Proving to Problem Solving" Proc. International Joint Conf. Artifical Intelligence Washington, 1969, pp.219-240.

Grosso74 Grosso, Franco, "Applications of Pattern Matching to Automatic Program Synthesis", Thesis, University of Torino, Italy, 1974 (in Italian).

Hoare73 Hoare, C. A. R.; Wirth, N., "An Axiomatic Definition of the Programming Language PASCAL", Acta Informatica, 2, (1973), pp. 335-355.

Karger74 Karger, P. A., 2nd Lt.; Schell, R. R., Maj., "MULTICS Security Evaluation: Vulnerability Analysis", ESD-TR-74-193, Vol. II, Electronics System Division, Hanscom AFB, Bedford, MA., June 1974.

King75 King, J. C., "A New Approach to Program Testing", Proc. 1975 International Conference on Reliable Software, April 21-23, 1975.

LaPadula73 LaPadula, Leonard J.; Bell, D. Elliot, "Secure Computer Systems: A Mathematical Model" MTR-2547, Vol. II The MITRE Corporation, Bedford MA, May 1973.

Levitt72 Levitt, K. N., "The Application of Program Proving Techniques to the Verification of Synchronization Processes" Proc. FJCC 1972 pp. 33-47.

_____76 Levitt, K. N., personal communication, 1976.

Lipton75 Lipton, Richard J., "Reduction: A Method of Proving Properties of Parallel Programs" CACM 18, 12, Dec. 1975.

Luckham76 Luckham, David; Suzuki, Norisha, "Verification-Oriented Proof Rules for Arrays, Records and Pointers", Stanford Artificial Intelligence Memo, 1976.

Masur75 Masur, H., "On a Class of Geodesics in Teichmueller Space", Annals Math 102, (1975)pp. 205-221.

Moore73 Moore, J. Strother, "Computational Logic: Structure Sharing and Proof of Program Properties" DCL Memo No 68 Dept. of Computational Logic, School of Artificial Intelligence, University of Edinburgh, 1973.

Moses71 Moses, J., "Symbolic Integration: The Stormy Decade", Proc. Second Symposium Symbolic and Algebraic Manipulation, (ACM) Los Angeles, March 1971, pp. 427-440.

Nilsson71 Nilsson, Nils J., Problem-Solving Methods in Artificial Intelligence, McGraw-Hill, N.Y., 1971.

Parnas72 Parnas, D. L., "A Technique for Software Module Specification with Examples", Communications of the ACM 15, 5 (May 1972), pp. 330-336.

Reifer74 Reifer, D. J. "Computer Program Verification/ Validation/ Certification", TOR-0074 (4112)-5, Aerospace Corporation, May 1974.

Robinson75 Robinson L.; Levitt K. N., "Proof Techniques for Hierarchically Structured Programs" Stanford Research Institute, Menlo Park, January 1975.

_____75a Robinson, L.; Levitt, K. N.; Neumann, Peter G.; Saxena, Ashok R., "On Attaining Reliable Software for a Secure Operating System" Proc. International Conf. Reliable Software Los Angeles, April 1975.

Rush72 Rush, K. (Deputy Secretary of Defense), "Security Requirements for Automatic Data Processing (ADP) Systems" Department of Defense Directive 5200.28, December 1972.

Schorre75 Schorre, Val, "A Program Verifier with Assertions in Terms of Abstract Data", SP-3841, System Development Corporation, Santa Monica, CA, October 1975.

Uncapher75 Uncapher, Keith W., "A Research Program in Computer Technology: Annual Technical Report, May 1974-June 1975", ISI/SR 75-3, Information Sciences Institute, University of Southern California, 1975.

Waldinger69 Waldinger, R. J.; Lee, R. C. T., "PROW: A Step Toward Automatic Program Writing" Proc. International Joint Conf. Artifical Intelligence Washington, May 1969, pp.241-252.

Weyrauch74 Weyrauch, R. W.; Thomas, A.J., "FOL: a Proof Checker for First-Order Logic" Memo AIM-235 Artificial Intelligence Laboratory, Stanford University, September 1974.

Wirth75 Wirth, N., "An Assessment of the Programming Language PASCAL", IEEE TSE, 1, 2, (June 1975), pp. 192-198.

Wulf70 Wulf, W. A., et al, "BLISS Reference Manual", Dept. Comp. Sci., Carnegie-Mellon University, January 1970.

AUTOMATED TOOLS

A PHILOSOPHY AND APPROACH

BARRY PRESS
CHIEF TECHNOLOGIST
APPLIED SOFTWARE LABORATORY
TRW
DEFENSE AND SPACE SYSTEMS GROUP

OVERVIEW

TOOLS . . .

- INTRODUCTION . . . SOFTWARE TOOLS
- USEAGE . . . WHY AND WHEN
- APPLICATIONS . . . REQUIREMENTS AND DESIGN
  CODE AND TEST
- CATALOG . . . SAMPLING OF EXISTING TOOLS
- AVAILABILITY . . . ACQUISITION OR DEVELOPMENT
- BUILDING . . . WHEN AND HOW
- POTENTIAL . . .

SOFTWARE TOOLS: WHAT ARE THEY?

- KERNIGHAN AND PLAUGER (ADDISON WESLEY, 1976)
  - FILTERS
  - FILE HANDLERS
  - SORTERS
  - EDITORS
  - FORMATTERS
  - TRANSLATORS

  } PROGRAMMING AIDS

- CLASSICAL V&V
  - DYNAMIC ANALYSIS TOOLS
  - STATIC ANALYSIS TOOLS
  - INTERPRETIVE SIMULATION
  - RETEST TOOLS

  } TEST TOOLS

- CONTINUOUS V&V
  - PROGRAMMING AIDS
  - TEST TOOLS
  - REQUIREMENTS AND DESIGN TOOLS
  - SYSTEM INTEGRATION TOOLS

USAGE: WHY AND WHEN

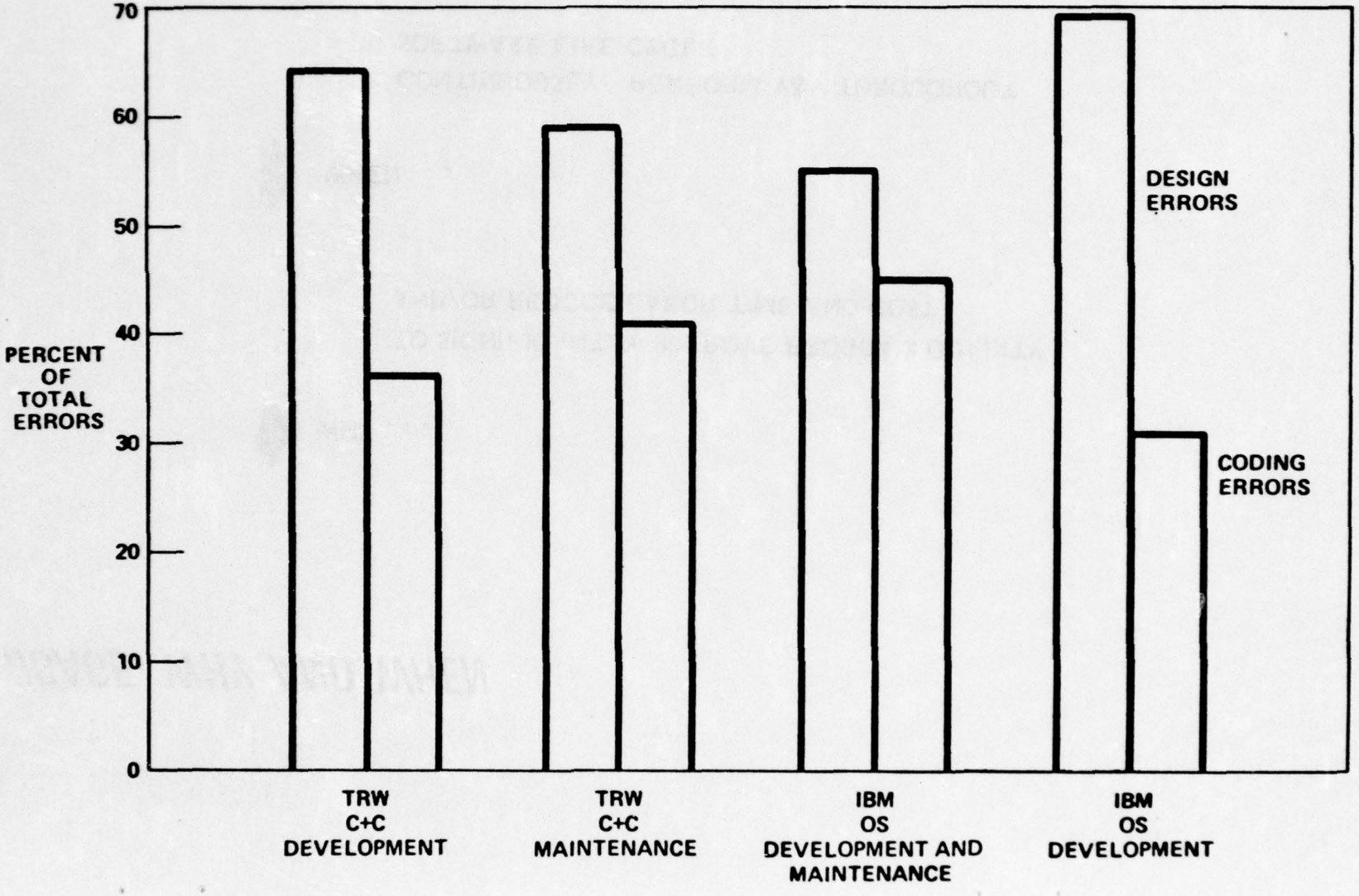
- WHY . . .

  TO SIGNIFICANTLY IMPROVE PROGRAM QUALITY
  AND/OR REDUCE LABOR TIME AND COST

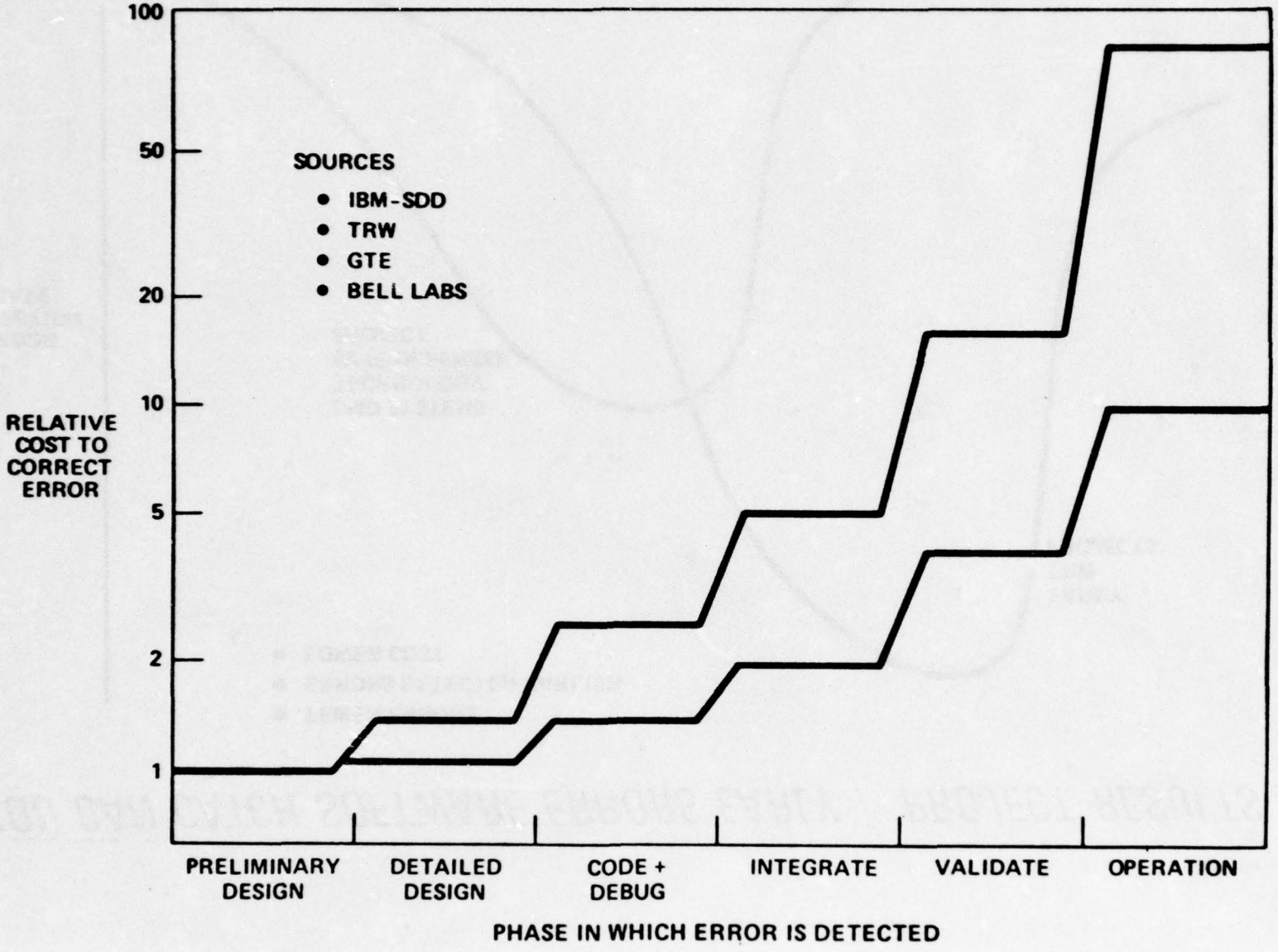
- WHEN . . .

  CONTINUOUSLY: PERFORM V&V THROUGHOUT
  SOFTWARE LIFE CYCLE

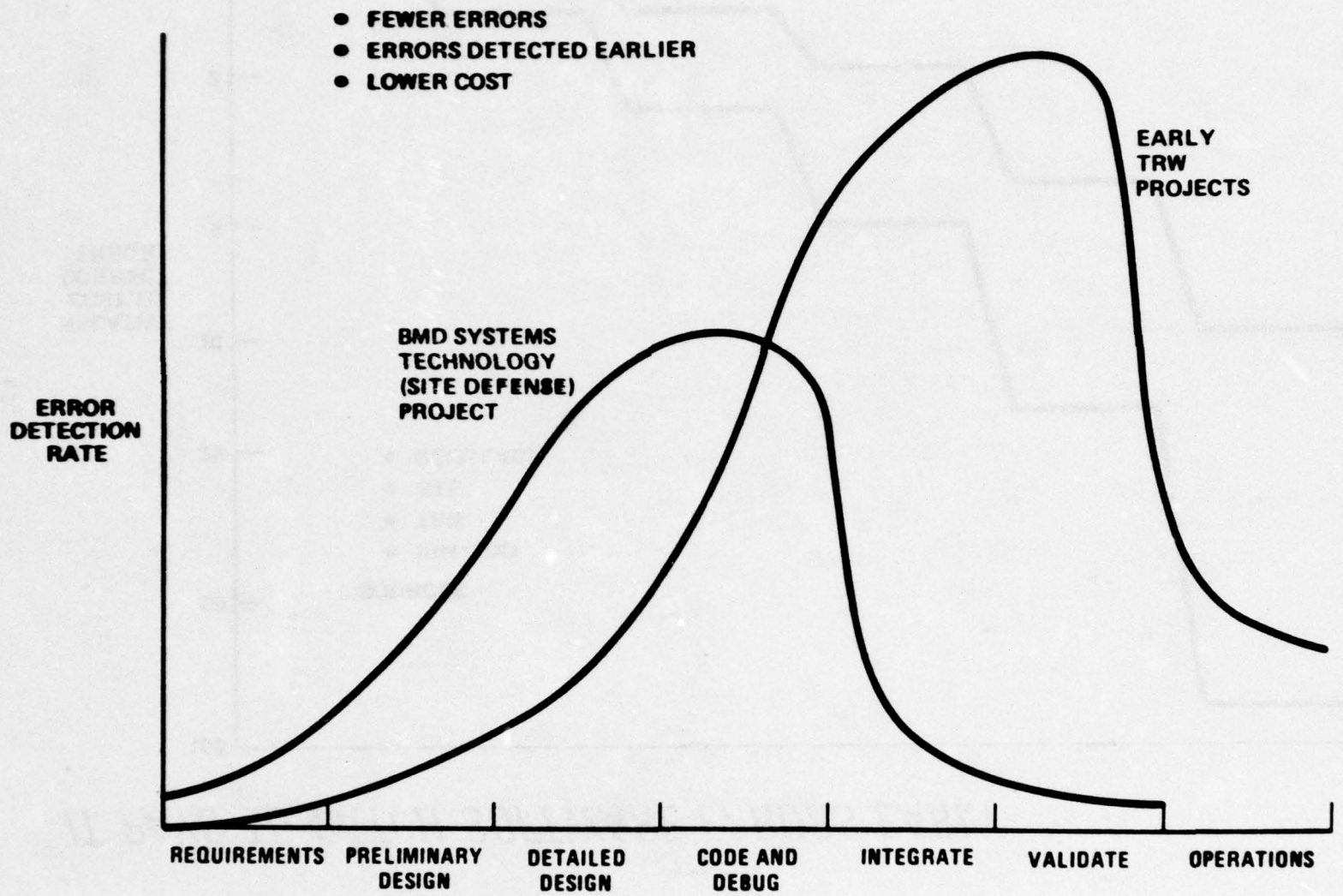
MOST ERRORS IN LARGE SOFTWARE SYSTEMS ARE IN EARLY STAGES



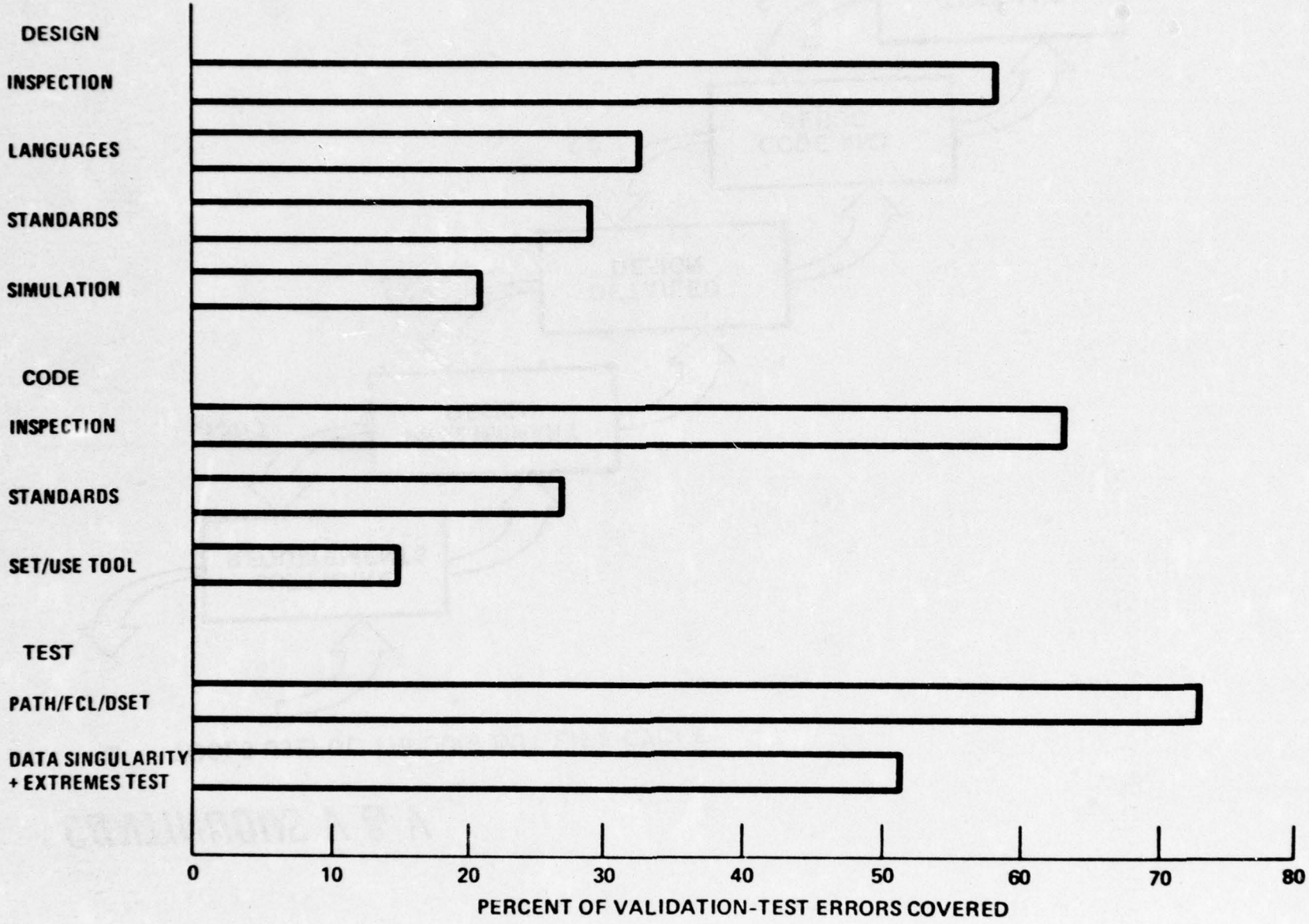
IT PAYS TO CATCH SOFTWARE ERRORS EARLY



YOU CAN CATCH SOFTWARE ERRORS EARLY : PROJECT RESULTS

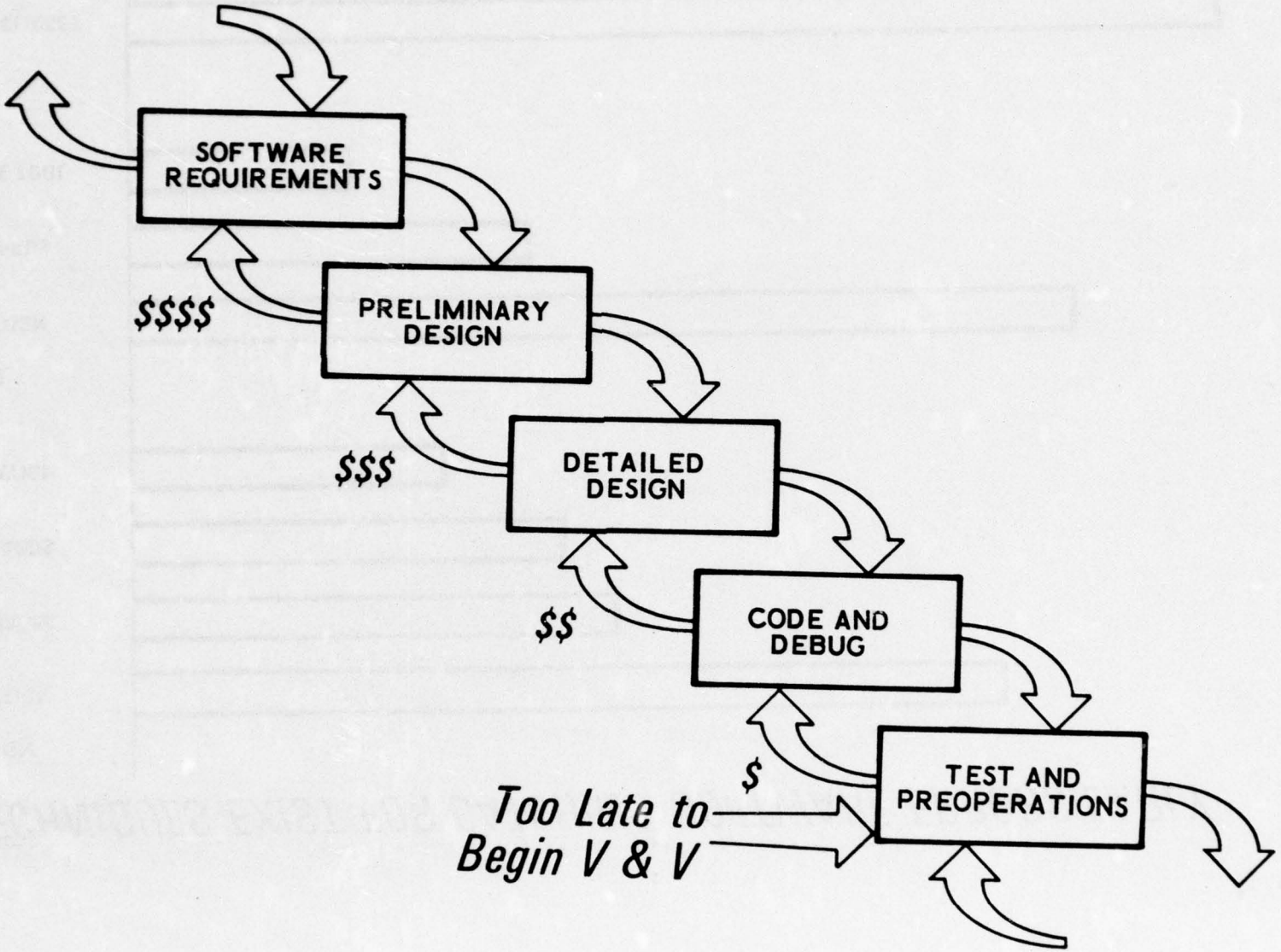


TECHNIQUES EXIST FOR CATCHING SOFTWARE ERRORS EARLY

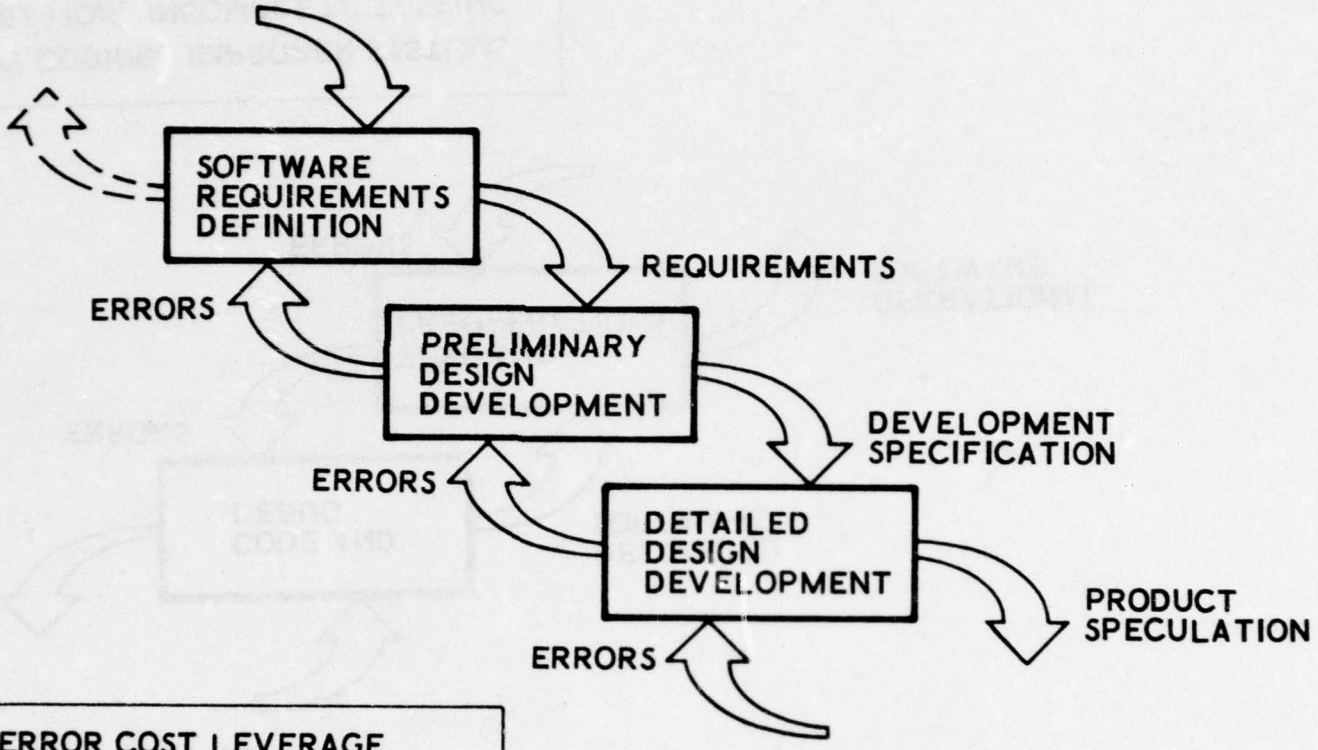


CONTINUOUS V & V

- TOOLS USEFUL THROUGHOUT LIFE CYCLE

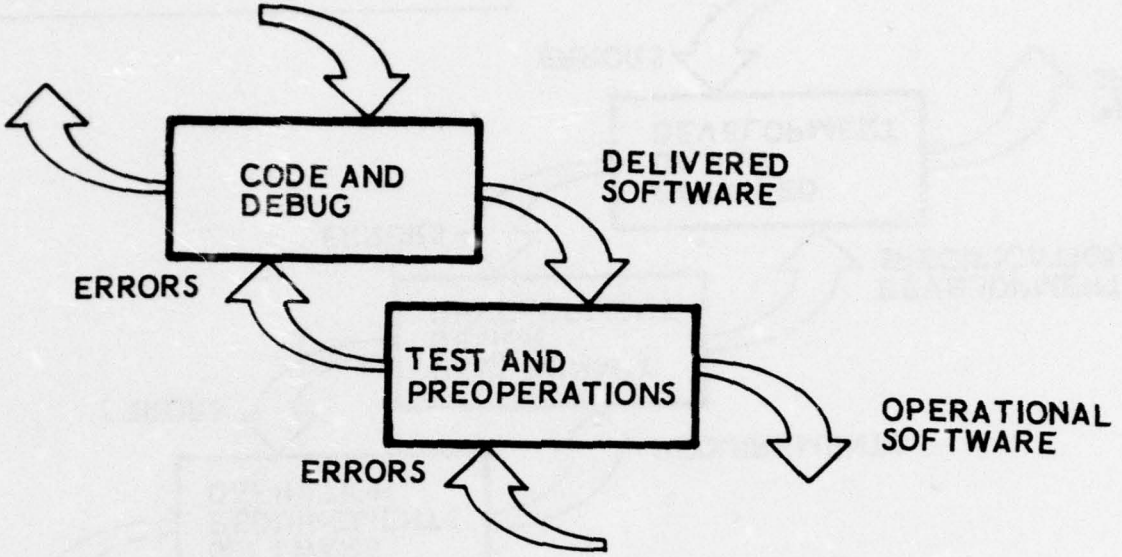


APPLICATIONS: REQUIREMENTS AND DESIGN



- HIGH ERROR COST LEVERAGE
- INCOMPLETENESS, INCONSISTENCY, OVERSPECIFICATION, AMBIGUITY
- "TESTING" DIFFICULT

APPLICATIONS: CODE AND TEST



- SLOPPY CODING, IMPROPER SYSTEMS INTEGRATION, INCOMPLETE TESTING, INADEQUATE AND INCOMPLETE RETEST
- SYSTEM RELIABILITY — "TRUST ME"

CATALOG: REQUIREMENTS AND DESIGN

- SOFTWARE REQUIREMENTS ENGINEERING
  - METHODOLOGY
  - REQUIREMENTS STATEMENT LANGUAGE
  - SUPPORT SOFTWARE
- PRELIMINARY DESIGN
  - PSL/PSA
  - DEVISE
  - CARA
  - CONSISTENCY CHECKER
- DETAILED DESIGN
  - PDL
- SIMULATORS

CATALOG: CODE AND TEST

- STATIC ANALYSIS
  - SURVAYOR
  - CODE AUDITOR
  - FLOP
- DYNAMIC ANALYSIS
  - NODAL
  - PACE
  - DIAGNOSTIC EMULATION
  - INTERPRETIVE COMPUTER SIMULATION
- SYSTEM ASSEMBLY AND INTEGRATION
  - COMPILERS
  - LINKAGE EDITORS
  - TEXT EDITORS
  - PROCESS CONSTRUCTION PROGRAM
- TEST ANALYSIS
  - MATCH
  - TDEM
  - RETEST

ACQUISITION

- SPECIFY REQUIREMENTS FIRST
- ENSURE COMPATIBILITY
- ACQUIRE EXISTING TOOLS
- DEVELOP TOOLS

BUILDING TOOLS

- WHEN . . .
  - NEW CAPABILITY
  - NEW LANGUAGES
  - NEW TECHNOLOGIES
- HOW . . .
  - WITH EXTREME FUNCTIONAL MODULARITY



TOOLS ARE Λ DELIVERABLE SOFTWARE: BUILD THEM THAT WAY

- REQUIREMENTS
- DESIGNS
- CODE AND DEBUG
- TEST
- CONTINUOUS V & V

BUILDING TOOLS: DIAGNOSTIC EMULATION

PROCESSING CONDITION	AVERAGE PROCESSING TIME (SEC)	SLOWDOWN FACTOR
TARGET COMPUTER INSTRUCTION PROCESSING	1.8	-
CDC 6600 ICS	66.1	37
VARIAN 73 DIAGNOSTIC EMULATION	16.2	9

- RESULT: 4:1 INCREASE IN PERFORMANCE
- 30:1 COMPUTER COST SAVINGS REALIZABLE DUE TO DIFFERENCES IN LARGE VERSUS MINI COMPUTERS

POTENTIAL

- PROOF OF CORRECTNESS
  - PROVING PROGRAMS
  - PROVING ASSERTIONS
- AUTOMATIC TEST CASE GENERATION
- TEST CASE LIBRARIES (COBOL, ALGOL 68)
- NATIONAL SOFTWARE WORKS
- FLEXIBLE ANALYSIS, SIMULATION, AND TEST FACILITY
- EXTENSIONS TO REQUIREMENTS AND DESIGN TOOLS

NEEDS OF VERIFICATION IN BALLISTIC MISSILE DEFENSE

MAJOR BEN A. JOHNSON
RESEARCH AND DEVELOPMENT COORDINATOR
DATA PROCESSING DIRECTORATE
BALLISTIC MISSILE DEFENSE ADVANCED TECHNOLOGY CENTER

INTRODUCTION.

Advances in the state-of-the-art of all technology disciplines surrounding the data processing subsystem in a Ballistic Missile Defense System ultimately depend to a large extent on the data processing subsystem for invocation and exploitation of those advances. For example, advanced sensors, both electromagnetic and optical, observing a ballistic missile attack would invoke a phenomenal real-time data base for processing through ultra-complex mathematical/logical algorithms. Battle durations are measured in seconds, responses in microseconds. A 30 second endoatmospheric engagement requiring and implemented on a 25 MIPS computing system implies that up to 750 million machine instructions must be discharged accurately and in proper sequence. The complexity, magnitude, and responsiveness required of a data processing system in such a role coupled with the fact that the only fully operational test possible is during a short stint at actual combat duty, imposes the emphasis that the BMD community must place on software testing, quality assurance, and reliability.

This paper only provides a brief overview of the problems being addressed by BMDATC research programs and a discussion of the basic approaches being pursued. More detailed information on the approaches is provided by the references shown in the bibliography.

The two major Verification and Validation (V&V) Programs currently being conducted by BMDATC are Adaptive Software Verification and Validation and Advanced Software Quality Assurance.

ADAPTIVE VERIFICATION AND VALIDATION.

The major objectives of adaptive verification and validation are to:

a. Provide better information regarding the performance limits of BMD software.

b. Significantly reduce the cost of testing BMD software to a desired level of confidence.

The former is achieved by determination of the actual limits of BMD processor performance as opposed to establishing adequate performance within some predefined envelope of threats. The latter objective is to be gained primarily through the introduction of sophisticated automation into the testing process in order to reduce tedious and time consuming manual activities.

Previous research in BMD software testing has concentrated on the problem of furnishing a consistent environment to all levels of the developing software and in providing a sufficiently realistic representation to the final software capable of at least limited real-time testing. The problems associated with the high costs of testing have been attacked through the earlier detection of errors when corrections are less complex and less likely to involve extensive retesting. It is still the case that performing a single test of an analytic simulation of the software on the actual real-time software is an expensive process. At this level of detail, the specification of the threat parameters as well as the parameters which give a sufficiently high-fidelity representation of the remaining BMD subsystems is a time-consuming, tedious process taking up to two weeks of man-time. Even more time-consuming is the analysis of the test results to see if performance was satisfactory. While performance in system terms--e.g., the total number of penetrators--is found relatively easily, evaluation of the performance of the various subfunctions of the data processing subsystem involves the examination of hundreds of parameters in combination as a function of time during the engagement. Experience with systems such as SAFEGUARD showed that many man-months of effort could be involved in analyzing a complex test.

Over and above the problems associated with defining a specific test scenario and analyzing the output is the problem of selecting the next test to run, particularly if it is to accomplish a very specific purpose such as placing additional load on a specific software function. In some cases the appropriate change is obvious, possibly just involving an increase in attack size. In other cases the action of the adaptive portions of the software itself, such as resource allocation, tend to obscure specific cause and effect vis-a-vis the threat scenario. Under these circumstances, a process of trial and error is often needed to achieve the desired effect.

When these considerations are placed in conjunction with the relatively large number of tests generally required to build confidence in the software, it is not surprising that the cost of testing is so high. Estimates for the broad spectrum of military software tend to put testing costs at around forth percent of the total cost. For BMD systems with their complexity and requirements for high confidence at deployment the fraction is probably higher.

The major functional components of a hypothetical adaptive tester being developed by BMDATC are shown in Figure 1.1. The role of each unit is as follows:

a. Automated Scenario Generator (ASG): This component accepts data from the analyst or from the Parameter Perturbation Algorithm and prepares the working data base containing all the information required for the development of a complete scenario by the Interactive Testbed. In interfacing with the analyst, particularly during initialization of an adaptive testing cycle, an interactive graphics capability is contemplated.

b. Interactive Testbed (ITB): The Interactive Testbed interfaces with the data processor or software under test, receiving output from it and modifying the simulated environment accordingly in order to provide a realistic data stream as input.

c. Performance Evaluator: This component accepts data recorded during testing from both the ITB and object under test. The data is then reduced according to predefined rules to produce performance measures.

d. Parameter Perturbation Algorithm (PPA): This function forms the "brain" of the Adaptive Tester. Reacting to the observed performance of the test object, the PPA determines the values to be assumed by the parameters which define a scenario for the next test and passes them to the scenario generator. The selection criteria are designed to efficiently drive the test object to the point at which "failure" occurs.

ADVANCED SOFTWARE QUALITY ASSURANCE.

The objectives of this project are:

a. To develop techniques for constructing verifiable BMD software.

b. To define tools and techniques for proving that BMD software is consistent with requirements.

c. To refine methods for implementing fault-tolerance in BMD software.

The current technique for validating software performance is testing. Testing techniques have a limited capability for validating compliance with requirements and provide an indication of performance boundries. The extreme size of BMD software and the number of combinations and permutations in the input space prevent even approaching a complete test of the software. In addition, singularities in software performance such as errors caused by incorrect logical sequences, erroneous variable substitution, or machine overflow and underflow are often only detected by accident. During the past

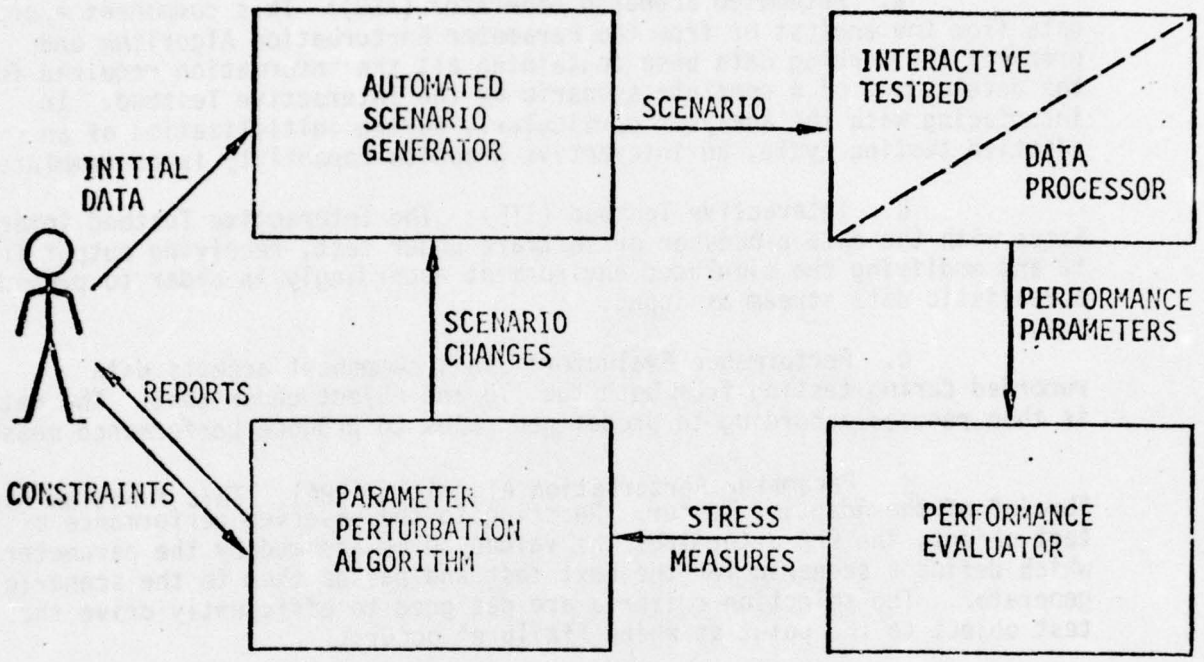


Figure 1.1. The Major Functional Components of an Adaptive Tester

five to ten years a large amount of theoretical research has been conducted on techniques of assuring program correctness, largely by academic theoreticians. This research has ranged from total automatic theorem proofs, which proves a program correct, to techniques of symbolic execution, which provide an algebraic equation that describes the operation of the code. This research has been restricted to small programs and has in general excluded a number of real world requirements such as real numbers, real-time operation and concurrency of operations.

The BMDATC research in advanced quality assurance is designed to extend these powerful software verification techniques to BMD software. The specific tasks to be performed under this effort include development and application of an executable assertion language, development of experimental preprocessors, and implementation of a software quality laboratory.

An assertion language will be designed which states intended input/output usage of program variables and describes relationships among program variables upon entry and exit from a module. Definition of a convenient-to-use syntax which expresses these properties will be the first issue addressed. Further research issues include extending the assertion language to enable proof-of-correctness and fault-folerance techniques to be applied in BMD software development. The initial definition of the assertion language will be an appropriate extension to FORTRAN-or PASCAL-based languages. Enhancements to the assertion language may depend on language features (such as user-defined data types) available in PASCAL but not in FORTRAN.

An experimental verification condition generator will be implemented which accepts, as input, programs annotated with entry/exit assertions, provides interactive assistance in generating loop invariant assertions, and automatically maps fully asserted programs into verification conditions (logical formulas suitable for theorem proving analysis). Programs written in a subset of FORTRAN or PASCAL will be processable.

Preprocessors will be implemented which will translate assertion language extensions to FORTRAN and PASCAL into statements acceptable to the base language compiler. The preprocessor approach was chosen in order to minimize implementation effort. Implementing the translation of executable assertion language input, output, entry, exit, and assert statements into executable FORTRAN statements is the initial goal of this element. A later goal is to implement a similar translation into executable PASCAL statements.

The software quality laboratory will consist of implementations of techniques for checking the consistency of executable assertion statements within software modules. The specific techniques to be implemented test structural characteristics, physical units, module interfaces, and clean termination. A major goal of this effort is to quantitatively assess the enhanced error detection capability provided by these consistency analysis techniques during a BMD software implementation using assertion statements.

BIBLIOGRAPHY

1. BMDATC Software Development System, Vol I, "Program Overview," BMDATC, July 1975.

2. BMDATC Software Development System, Vol II, "Research Description," BMDATC, July 1975.

3. Software Verification and Validation Technology Final Report, Vols. I and II, General Research Corporation CR-7-626, February 1976.

4. Hierarchical Testing Methodology/Capability Description, General Research Corporation CR-2-626/R1, November 1975.

5. Hierarchical Testing Methodology Capabilities Description, General Research Corporation CR-2-626/R1.

6. Research Plan for Adaptive V&V, General Research Corporation CR-1-708, May 1976.

7. Research Plan for Reliable Software, General Research Corporation CR-1-676, November 1975.

8. Final Report for Reliable Software, General Research Corporation CR-2-676, December 1975.

Computer Program Verification and Validation

MR. DONALD J. REIFER
The Aerospace Corporation

DOD/Industry Invitational Conference
on Software Verification and Validation

3-5 August 1976

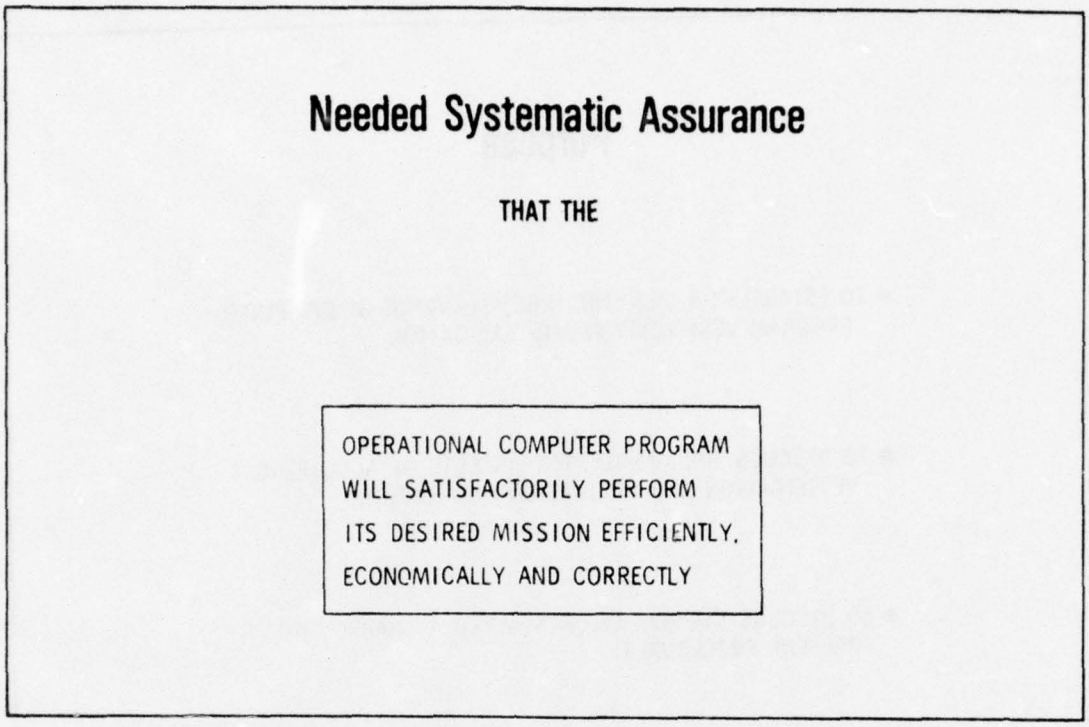
For further information concerning this presentation contact:

Mr. Donald J. Reifer
Space Transportation Directorate
Vehicle Systems Division
The Aerospace Corporation
P.O. Box 92957
Los Angeles, CA 90009
(213) 648-5890

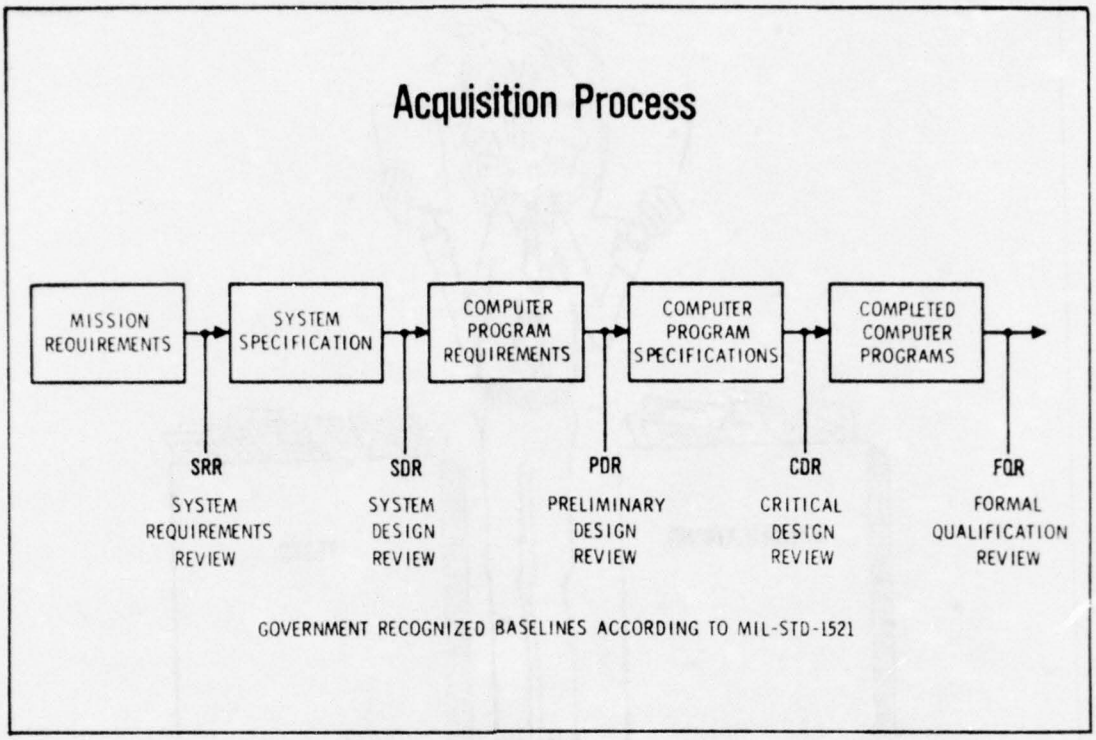
Purpose

- TO ESTABLISH A WORKING UNDERSTANDING OF COMPUTER PROGRAM VERIFICATION AND VALIDATION
- TO DISCUSS THE MANAGEMENT ASPECTS OF ACQUIRING A VERIFICATION AND VALIDATION CONTRACTOR
- TO DISCUSS CURRENT EFFORTS AIMED AT IMPROVING PRESENT PROCEDURES

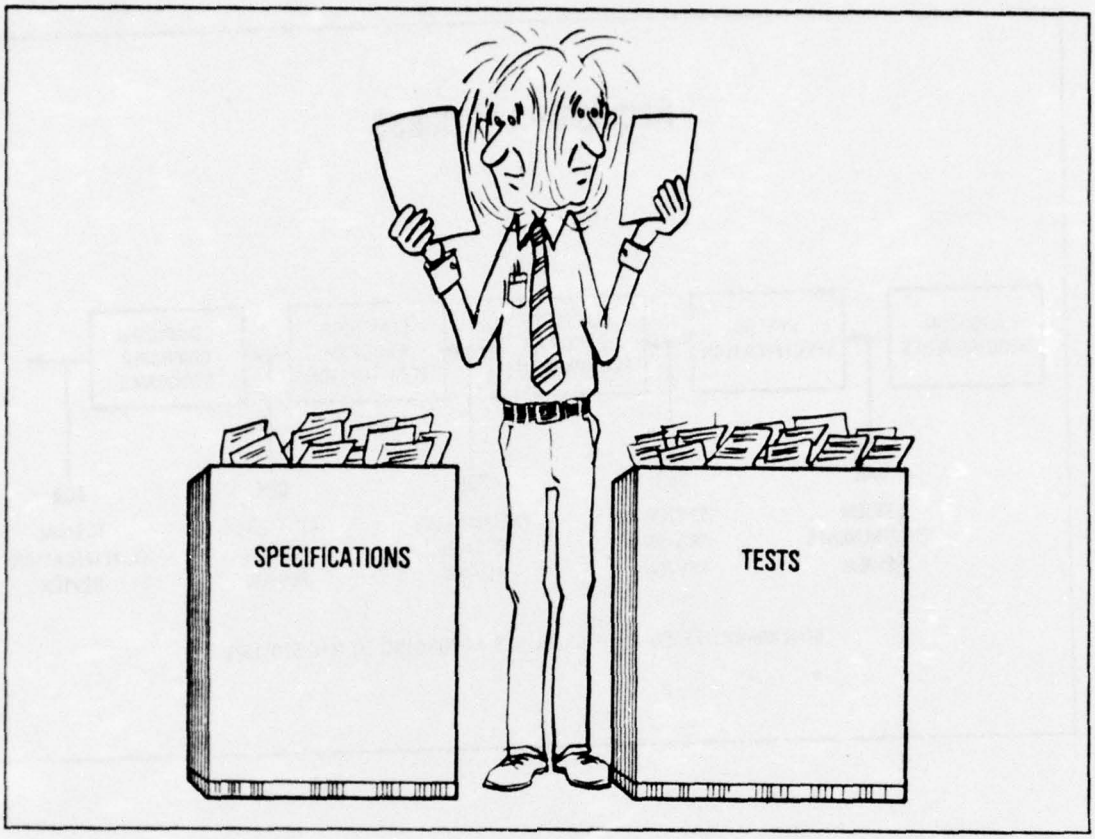
The purpose of this presentation is to describe the techniques of computer program verification and validation and explain how they can be successfully employed to improve the quality of weapons system software. In order to achieve this objective, we shall first establish a working understanding of computer program verification and validation as it is practiced today. Then, we shall discuss management practices and procedures for acquiring the services of an independent contractor. Finally, we shall explain our current work aimed at improving the state-of-the-art.



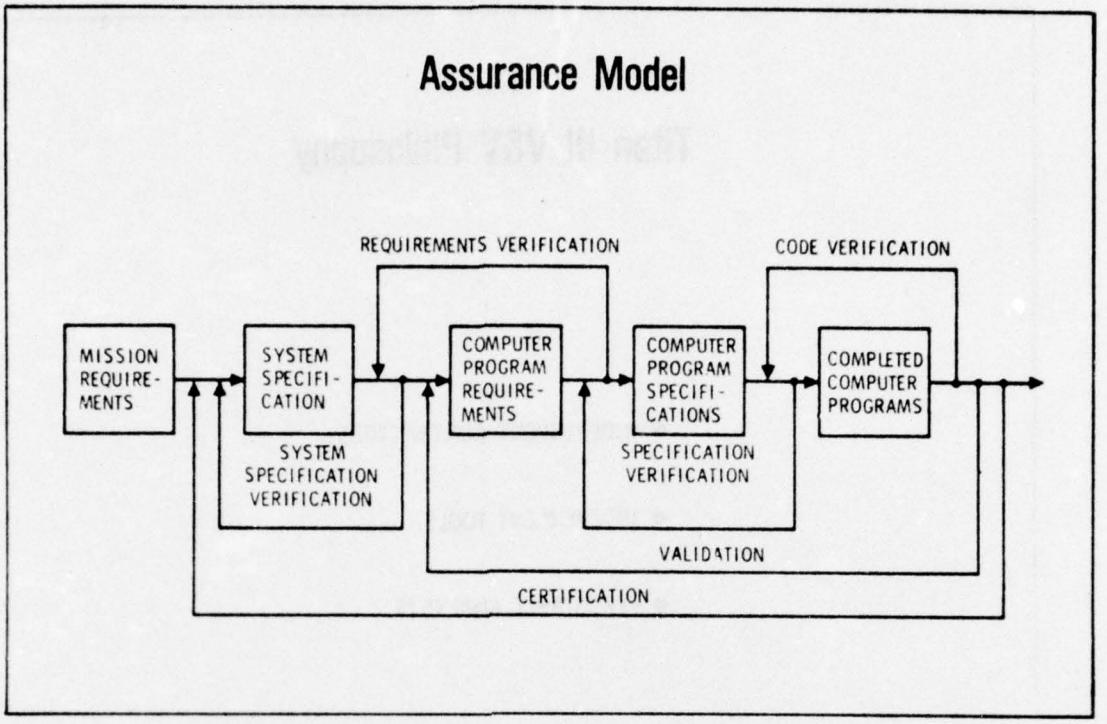
The Space and Missile Systems Organization (SAMSO) of the U.S. Air Force does not develop its own computer software. Instead, it depends on contractors to provide it as part of the weapons systems it acquires. This dependence has created problems more related to the methods it uses for acquisition management than those it uses for software development. One of the most successful approaches applied by SAMSO Program Offices to cope with these acquisition management problems is computer program verification and validation. Verification and validation are techniques used to provide management with systematic assurance that the computer programs they acquire will perform their mission requirements economically, efficiently, and correctly.



Software acquisitions at SAMSO are developed using a sequential life cycle as defined in AFR 800-14, Volume 2 and other applicable military regulations. The purpose of so ordering the development is to create a series of validated baselines upon which the computer software products can be developed and tested. Typically, these baselines are documents which specify either software requirements or computer programs as actually built. Management of the sequential life cycle is accomplished by conducting a series of reviews and audits as described in MIL-STD-1521. The purpose of these meetings is two-fold. First, they provide management with an assessment of the project status. Then, they are used to pinpoint technical problems that require attention and/or action.



Reviews are preceded by an examination of pertinent documents and/or code. The actual masses of paper produced to describe the computer program, its implementation, and the tests that qualify it are enormous. More often than not, sufficient time and/or resources are not available to do an adequate job. What results is a review that uncovers inconsistencies. The technical basis and decisions upon which the design is founded are frequently not challenged. More and better feedback is required throughout the development cycle to provide management with assurance that their computer software will perform as specified and will meet all of its design goals.

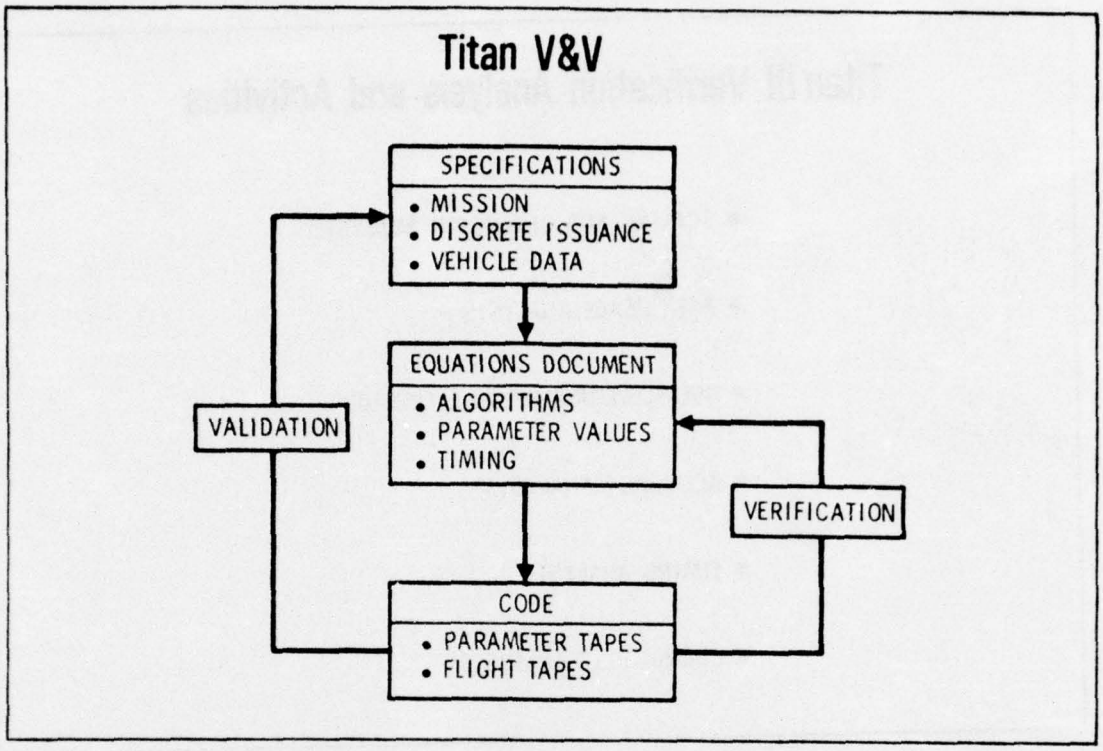


Computer program verification and validation provide management with the feedback they require to assess the integrity of their software products. Computer program verification is the iterative process of continuously determining whether the product of each step of the computer program acquisition process fulfills all requirements levied by the previous step, including those set for quality. Computer program validation is the test and evaluation of the complete computer program aimed at ensuring compliance with the performance and design criteria. Computer program certification is the test and evaluation of the complete computer program aimed at ensuring operational effectiveness and suitability with respect to mission requirements under realistic operating conditions.

Titan III V&V Philosophy

- INDEPENDENT CONTRACTORS
- INDEPENDENT TOOLS
- EXECUTABLE ANALYSIS

The Titan IIIC was one of the first projects to use computer program verification and validation. Titan is an expendable booster used to place satellite payloads into orbit. This launch vehicle uses an embedded computer system for guidance and issuing discretes. The flight program consists of guidance equations, digital flight control equations and interrupts, interface logic, self-test, sumcheck, and telemetry formatting routines. Contractors involved in producing the flight program are the Aerospace Corp. (guidance parameters), Martin Company (reference trajectory, pretest trajectory, and range safety) and Delco Electronics (IMU and parameter tapes). Each of these contractors independently checks the other's work using independent sets of support tools to ensure proper implementation.



The two efforts that are used to flight certify the Titan IIIC computer programs are verification and validation. Validation is a gross test to demonstrate that the equations as coded on the flight tapes meet all specifications; whereas, verification is a detailed test to assure that the tapes are coded in accordance with the equations document. Although the goal of validation and verification are the same, the role each plays is vastly different. Validation consists of overall performance testing to demonstrate the adequacy of the equations as coded under simulated flight conditions. On the other hand, the intent of verification is not to question validity, but rather to check in detail if the flight tapes are coded and punched in conformance with the equations document.

Titan III Verification Analysis and Activities

- SCALING AND DIMENSION ANALYSIS
- PATH USAGE ANALYSIS
- OVERFLOW/UNDERFLOW EXECUTION
- ACCURACY ANALYSIS
- TIMING ANALYSIS
- CODING RULE ANALYSIS

The purpose of verification is to ensure that the computer program completely and correctly meets every requirement of the Program Requirements Document. Verification is accomplished by a process of detailed tests which demonstrate on a step by step basis that the results of executing every equation, logic branch, and input/output statement faithfully satisfy the requirements of the specification. The Titan IIIC computer programs are tested for:

1. Correctness: performance of the algorithm and/or logic as specified.
2. Continuity: execution of specified transfers and selection of proper branch destinations.
3. Overflow: generation of an overflow or improper divide for a correlated set of data.
4. Accuracy: generation of a result that is accurate to within the allowable least significant bit.

Titan III Verification Tools

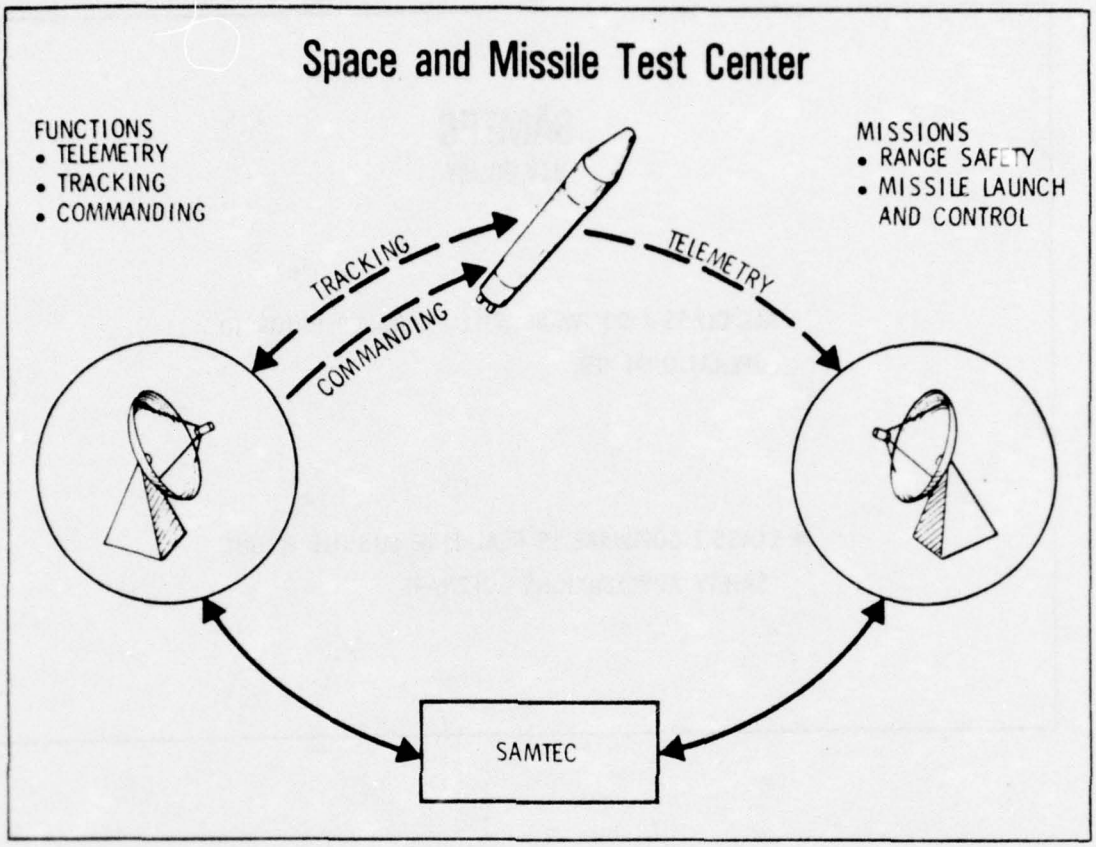
- OPEN-LOOP AUTOMATIC ANALYZER (Martin)
  - EXECUTES CODE IN FLIGHT CONFIGURATION COMPUTER
  - COMPARES RESULTS TO SCIENTIFIC SIMULATION RESULTS
- CODE COMPARATOR (Aerospace)
  - EXECUTES CODE IN INTERPRETIVE COMPUTER SIMULATOR
  - COMPARES RESULTS TO SCIENTIFIC SIMULATION RESULTS

Titan IIIC verification tools include the Martin Company-developed, open-loop, automatic analyzer and the Aerospace Corporation-developed code comparator. The difference between the two tools is that the analyzer uses the actual flight computer, while the comparator simulates its execution characteristics using an interpretive computer simulator. In either case, the flight code is executed and compared, statement-by-statement, with an independently coded FORTRAN version of the program. All miscomparisons are automatically flagged. To pass the success criteria, arithmetic differences between the two versions must be within specified tolerance levels. Integers, logic branches, and input/output are also checked and must be absolutely as specified.

Titan III Validation Techniques

- ICS VALIDATION
  - RUN CASES WITH NOMINAL AND DISPERSED DATA
  - COMPARISON OF SCIENTIFIC vs ICS RUNS
  - COMPARISON OF RUN BETWEEN CONTRACTORS
  - IMAGE RESTARTS TO FORCE BACKUP PATHS
- HARDWARE VALIDATION
  - MARRY FLIGHT CONFIGURATION HARDWARE MOCKUP TO ANALOG SIMULATION
  - COMPARE TELEMETRY DATA TO SPECIFICATION AND ICS RUNS

Titan IIIC validation is conducted to demonstrate that the equations, as coded, properly perform and that the mission specifications are met under simulated flight conditions. Aerospace makes an interpretative computer simulation (ICS) run using parameter tapes for the initial part of the boost trajectory. By comparing the resultant output data from the ICS trajectory run against equivalent counterparts in a scientific trajectory run, Aerospace is able to test on a system basis whether the parameter tapes perform as expected. Delco and Martin also validate by making independent, six-dimensional, scientific trajectory runs to test the suitability of the guidance parameters in meeting specifications. Other techniques used for Titan mate hardware to analog and telemetry devices. These are used to provide additional data against which the tapes can be validated.



Another approach to computer program verification and validation is practiced at the Space and Missile Test Center (SAMTEC). SAMTEC operates the Western Test Range, a widely dispersed network of radar and optical sensors, communication links, and data processing facilities. Its basic mission is to provide support to both ballistic missile and satellite programs. Facilities along the California coast provide optical and radar tracking of the missile or space booster, transmit commands, and receive telemetry during the launch phase of flight. Facilities in the Hawaiian Islands support missile flights in mid-course, while fixed, airborne, and shipborne facilities provide the instrumentation in the terminal areas.

SAMTEC
V&V POLICY

- ALL CLASS 1 SOFTWARE WILL BE V&V'ED PRIOR TO OPERATIONAL USE

- CLASS 1 SOFTWARE IS REAL-TIME MISSILE FLIGHT SAFETY APPLICATIONS SOFTWARE

SAMTEC has established the policy that all Class 1 software will be verified and validated by an independent contractor prior to being placed in operational use. Class 1 software encompasses computer programs written for real-time missile flight safety applications. The potential dangers to life and property warrant the additional degree of protection afforded by this approach. However, it is also necessary not to incorrectly abort a good missile. Therefore, software systems at SAMTEC must support missile safety officers in achieving both these goals.

Areas of Activity for V&V Contractor

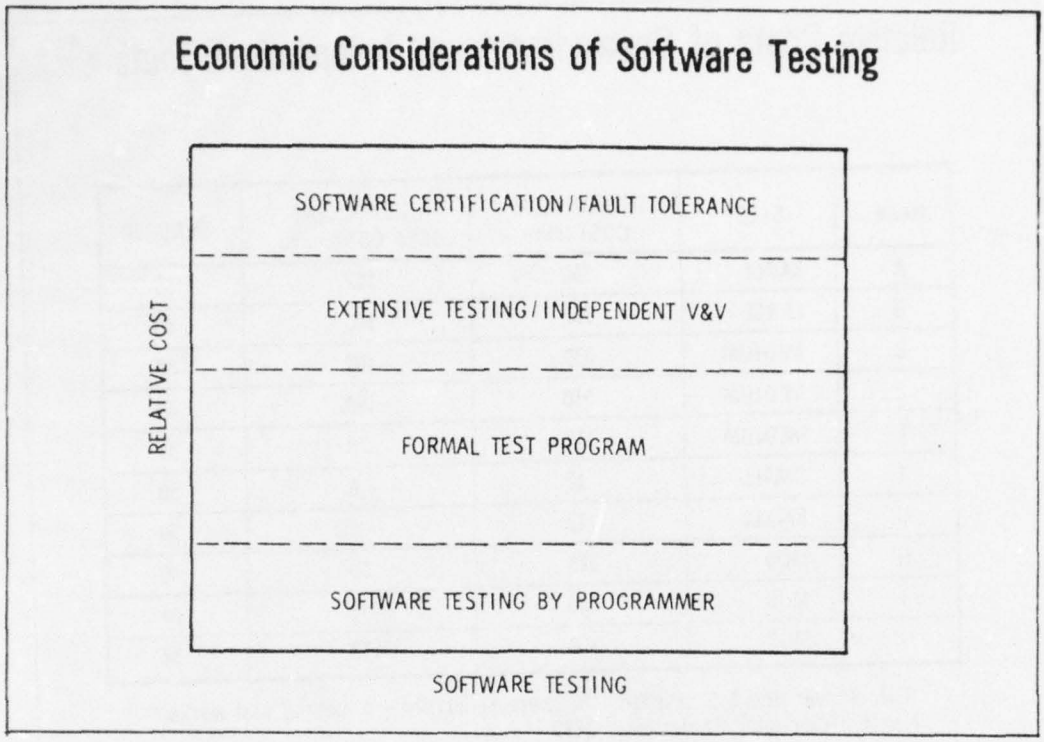
- NEW SOFTWARE
  - SPECIFICATION REVIEW
  - ANALYSIS AND TESTING
- MODIFIED OPERATIONAL SOFTWARE
  - SUBSTANTIVE MODS
  - NONSUBSTANTIVE MODS
- GENERATION OF V&V TOOLS
  - PROPOSAL AND SCREENING
  - PREPARATION
- SPECIAL STUDIES
  - QUICK REACTION
  - CRITIQUES AND TRAINING

SAMTEC has acquired the services of an independent verification and validation contractor to help them achieve these goals. Contractually, the services provided by this contractor include participating in reviews, studies, and analyses and designing and conducting tests aimed at ensuring that critical requirements have been met. The results of using this approach have been encouraging. As an example, a 25,000 word program that was an integral part of the range safety system at Vandenberg AFB for the last eight years was evaluated. Twenty major errors were detected, seven of which were critical to range operations. In another case, a new range safety system was evaluated. It was determined to be unready for operational use and subsequently had to undergo considerable modification.

V&V Contractor Activities

	SDR	PDR	CDR	FQR
DOCUMENT REVIEW	X	X	X	X
REQUIREMENTS ANALYSIS	X	X		
V&V PLAN		X	X	
ALGORITHM EVALUATION			X	
TOOL PREPARATION			X	
SPECIAL STUDIES	X	X	X	X
CODE ANALYSIS				X

Based on our Titan IIIC, SAMTEC, and other project experiences, it seems that there is much an independent verification and validation contractor can do. The verification and validation contractor should support document reviews and special studies. He should always be tasked to identify critical requirements and formulate a verification and validation plan. In this way, his expertise can be used to provide the detailed technical feedback we need to judge the soundness of the design and the integrity of the resulting products. As funding permits, the independent contractor can analyze key algorithms and provide recommended improvements or alternatives. He can prepare test tools and be tasked to independently test and evaluate the code to ensure proper compliance. He can even be tasked to prepare specifications.



Software testing often accounts for 40 to 50 percent of the computer program development budget. In spite of this large investment, it is generally agreed that the results of testing are uncertain. Therefore, the Air Force uses other approaches to increase its confidence in its final software products. At SAMSO, our reliability requirements are so stringent that we have to apply such test techniques as independent verification and validation and fault-tolerance. These approaches cause us to expend additional funds during development. These funds can be significant. Yet, they are warranted because we have only a limited opportunity to recover from a failure once the satellite or missile is launched.

Relative Costs of Development and Independent Testing*

NAME	SIZE	DEVELOPMENT COST (mm)	INDEPENDENT TEST COST (mm)	PERCENT
A	LARGE	750	252	33
B	LARGE	750	300	40
C	MEDIUM	337	142	42
D	MEDIUM	310	150	48
E	MEDIUM	112	38	33
F	SMALL	12	6	50
G	SMALL	12	6	50
H	MOD	125	50	40
I	MOD	650	450	69
J	MOD	300	112	37

* R.H. Thayer and E.S. Hinton, "software reliability - a method that works," National Computer Conference, 1975

The concept of test and evaluation by an independent contractor implies an additional cost of some magnitude. Based on data from ten different projects, these costs range from 33 to 69 percent of that expended on software development. The higher costs imply a considerable analysis effort, a heavy use of specially constructed software tools, and a comprehensive test and execution analysis of the completed code. The lower costs are associated with a monitoring activity which has limited test and execution analysis coupled to it.

Lessons Learned

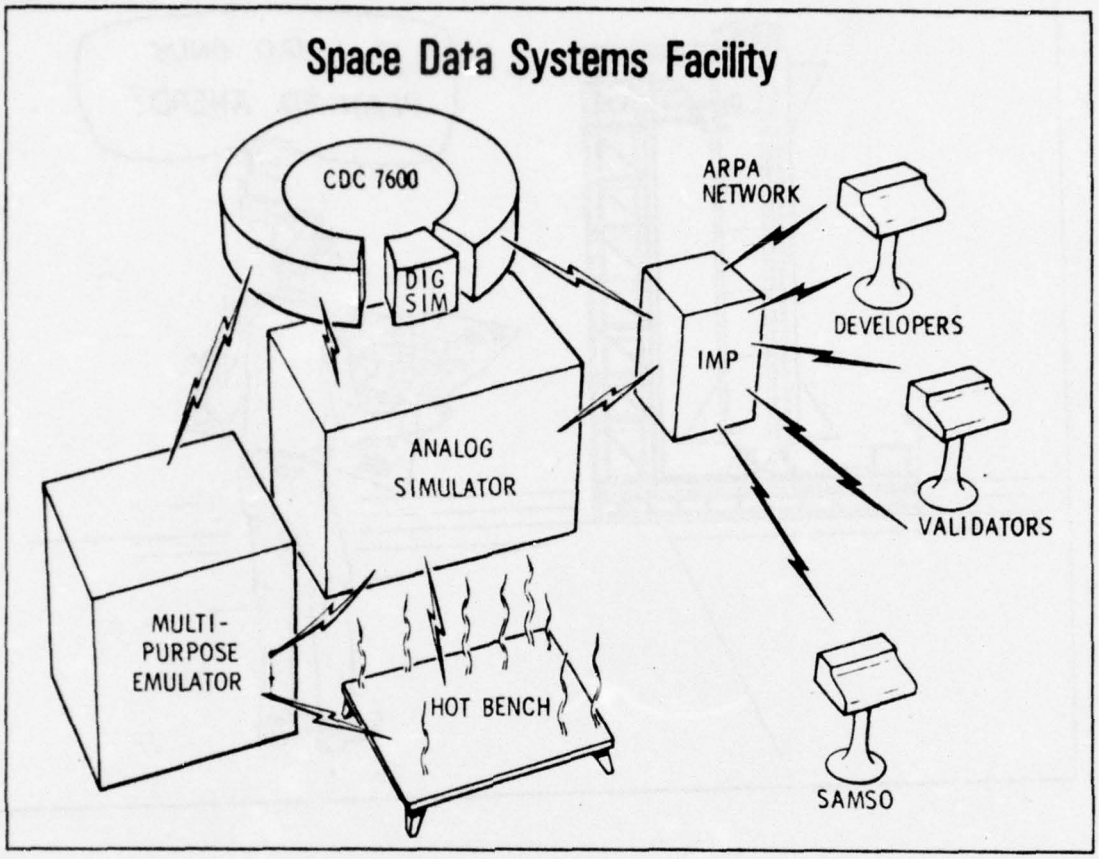
- V&V MUST BE TAILORED TO EACH PROJECT USING IT
- KEY V&V TO CRITICAL AREAS AND SIGNIFICANT DISCREPANCIES
- V&V HAS BEEN SUCCESSFUL - BUT ONCE INITIATED, MUST CONTINUE
- MAINTAIN CLEAR ACCOUNTABILITY FOR RESULTS
- USE EXPERIENCED PEOPLE
- INTERFACE PROBLEMS ARE THE MAJOR DIFFICULTY
- COSTS CAN BE REDUCED USING AUTOMATED TOOLS

Our experiences in applying computer program verification and validation have taught us some useful lessons. First, concentrate on the critical requirements and the significant discrepancies because these are the ones that it is cost effective to prove or remove. Second, understand that whenever there are multiple organizations working on a project there are going to be interface problems. Direct your attention to them early and try to solve them by making contractors accountable and by using experienced people. Third, realize that costs can be reduced using automated tools. Make your contractors justify the economics of their use, though, before you procure them. Fourth, remember that verification and validation must be continued once initiated to ensure success.

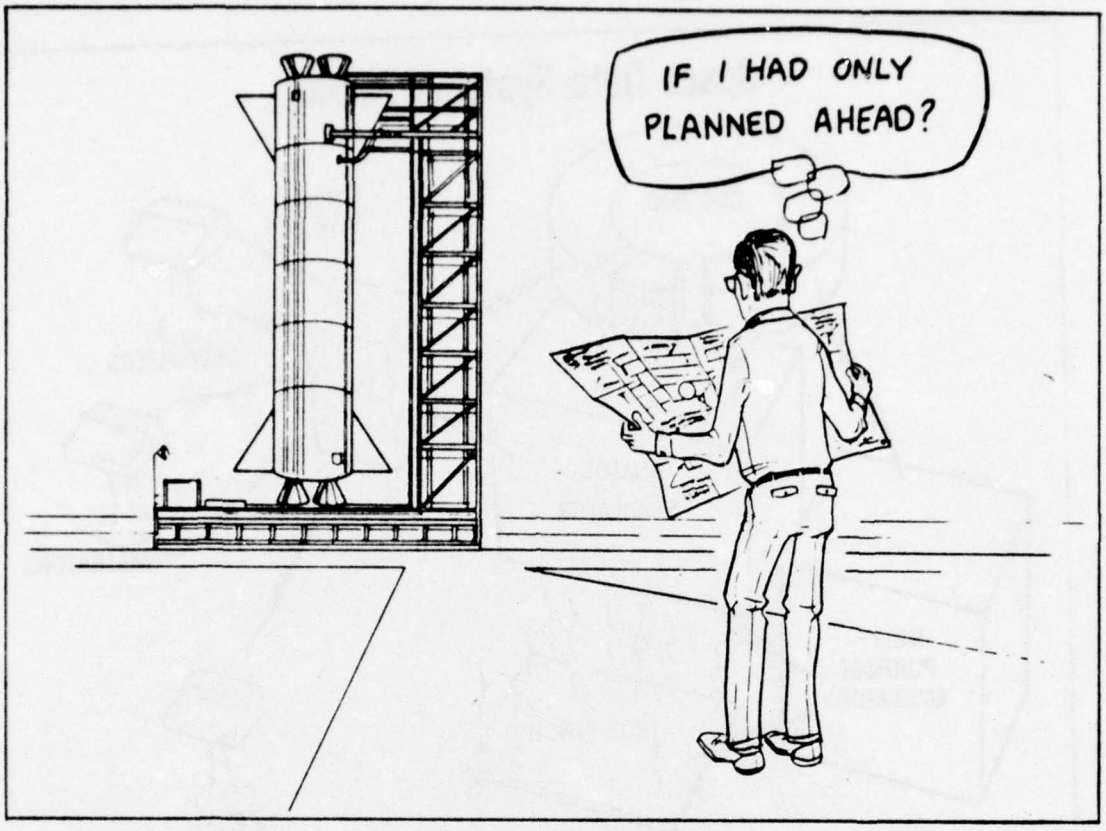
Management Control

- CONTRACT
- TASK APPROVAL
- WRITTEN REPORTS
- MONTHLY PROJECT REVIEWS
- INTERFACE CONTROL
  - ALL ONE TEAM
  - COMMUNICATIONS CHANNELS OPEN
  - EARLY INVOLVEMENT
- TOOLS DELIVERABLE

We have also learned to manage our verification and validation contracts better. We suggest that a fixed price, level of effort contract with task approval be considered during procurement. This type of contract enables the project officer to task the contractor to perform special studies and analyses as required. We suggest that if tools are developed as part of the contract, they be screened before their development is approved to see if they can be made generally applicable. If they can, we suggest they be delivered with appropriate documentation. Written reports for each task and informal, monthly management reviews are recommended. We also recommend that you endeavor to instill a team spirit and attack the interface problems early in the project.



SAMSO has initiated an advanced development program to create the Space Data Systems Facility (SDSF). The SDSF is a ground-based computational laboratory which will be used to develop and verify and validate flight programs (both computer programs and micro-programs) for a variety of space and missile applications. The project initiated this year will take five years to fully develop. An interim SDSF capability will be available in two years. The facility will be housed at SAMSO where it may be made available to both SPO and contractor users. The facility will have sophisticated emulation, simulation (both hybrid and digital), and diagnostic capabilities. A variety of standard software tools will be available as required.



The purpose of this presentation was to establish a working understanding of computer program verification and validation and to explain how SAMSO employs them to improve the quality of weapons system software. Verification and validation should be employed throughout the life cycle to provide technical feedback as to product status and quality. When properly applied, the techniques help uncover problems early so that they can be corrected. The cost of applying verification and validation to large projects could get very expensive if our launch vehicle experience serves as a valid model. To use these concepts, different strategies for their application need to be formulated based on available funding and assessments of technical risk and complexity.

Consistency Checker, Summary

Michael Landes
Rome Air Development Center

Mr. Michael Landes RADC/ISIM gave a presentation on the ongoing development of the Consistency Checker and the Computer Software Specification.

The Consistency Checker, an automated verification software tool which collects a file of pertinent data, imposes a methodology on the design phase personnel. This data file is designed compatible with CARA system output and is used to verify the design phase against the CARA output of the requirements phase.

The Computer Software Specification is a collection of sections properly worded for insertion in a software Statement of Work to ensure that elements of modern programming practices are mandated. This evolutionary vehicle is now under revision to form a joint RADC/ESD Computer Software Specification.

AMPIC

PRECEDING PAGE BLANK-NOT FILMED

MIKE IKEZAWA

LOGICON

AMPIC: SHORT DESCRIPTION

AMPIC IS A PROGRAM THAT

STRUCTURES,

TRANSLATES, AND

SYMBOLICALLY EXECUTES

OTHER PROGRAMS.

A complete AMPIC run on a given input module consists of four AMPIC phases. The input module can be in a higher-order language (like FORTRAN) or in an assembly language.

Useful results are obtained at each AMPIC phase except Phase I.

AMPIC PHASES

PHASE	FUNCTION	INPUT	OUTPUT
I	PREPROCESSING	INPUT MODULE	CODE SEGMENTS STRUCTURAL TABLE
II	STRUCTURING	STRUCTURAL TABLE	STRUCTURAL FORMULA FLOWCHART
III	TRANSLATION	CODE SEGMENTS	TRANSLATED SEGMENTS
IV	PATH ANALYSIS	STRUCTURAL FORMULA TRANSLATED SEGMENTS	SYMBOLIC EXECUTION RESULTS

AMPIC will be described by means of actual AMPIC output on two examples.

The first example (left side of next page) is a FORTRAN module that computes the (real) roots of quadratic equations using the quadratic formula.

First, AMPIC segments the module into code groups that can be treated as individual elements (nodes) of a flowchart. It creates its own "labels" using "Tn" for branching elements and "Sn" for non-branching elements. (Right side of next page).

AMPIC FORTRAN EXAMPLE: CODE PARTITIONING (PHASE I)

```
      SUBROUTINE ROOT
      IMPLICIT INTEGER*2(I,K-N)
      COMMON/ARGS/A,B,C
      COMMON/VALUE/ROOT1,ROOT2,NROOT
      IF(A.EQ.0) GOTO 100
      TEMP1 = 4.*A*C
      TEMP1 = B*B - TEMP1
      IF(TEMP1.LT.0) GOTO 100
      TEMP2 = 2.*A
      ROOT2 = -(B/TEMP2)
      IF(TEMP1.EQ.0) GOTO 300
      TEMP2 = FSQRT(TEMP1)/TEMP2
      ROOT1 = ROOT2 + TEMP2
      ROOT2 = ROOT2 - TEMP2
      NROOT = 2
      RETURN
100   NROOT = 0
      RETURN
300   ROOT1 = ROOT2
      NROOT = 1
      RETURN
      END
```

```
T1    IF(A.EQ.0.) GOTO S7

S2    TEMP1=4.*A*C
      TEMP1=B*B-TEMP1

T3    IF(TEMP1.LT.0.) GOTO S7

S4    TEMP2=2.*A
      ROOT2=-B/TEMP2

T5    IF(TEMP1.EQ.0.) GOTO S8

S6    TEMP2=FSQRT(TEMP1)/TEMP2
      ROOT1=ROOT2+TEMP2
      ROOT2=ROOT2-TEMP2
      NROOT=2
      GOTO OUT

S7    NROOT=0
      GOTO OUT

S8    ROOT1=ROOT2
      NROOT=1
      GOTO OUT
```

Next, AMPIC constructs a "structured" flowchart of the module in terms of the Tn and Sn (right side of next page). (The left side shows the previous result). This is a "structured" flowchart, because the flowchart conforms to the rules of "structured programming." Note, however, that the flowcharted module itself is not "structured." (If the statement "NROOT=0" occurred twice in the module itself, then the module could have been a "structured" module).

The flowcharting conventions at Tn are:

Down: No jump

Right: Jump ("go to")

The "100" at S7 and "300" at S8 are original statement labels.

AMPIC FORTRAN EXAMPLE: STRUCTURED REPRESENTATION (PHASE II)

```
T1   IF(A.EQ.0.) GOTO S7

S2   TEMP1=4.*A*C
     TEMP1=B*B-TEMP1

T3   IF(TEMP1.LT.0.) GOTO S7

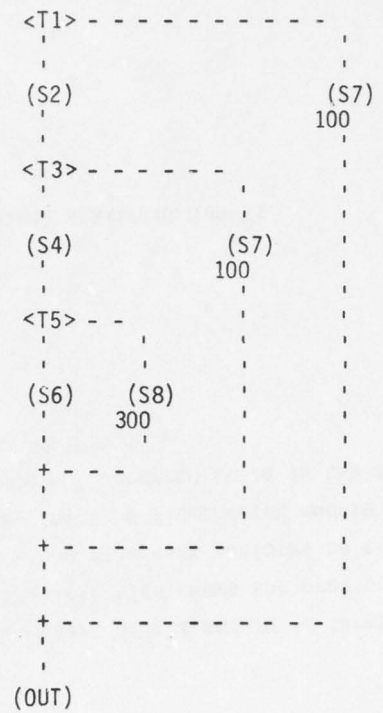
S4   TEMP2=2.*A
     ROOT2=-B/TEMP2

T5   IF(TEMP1.EQ.0.) GOTO S8

S6   TEMP2=FSQRT(TEMP1)/TEMP2
     ROOT1=ROOT2+TEMP2
     ROOT2=ROOT2-TEMP2
     NROOT=2
     GOTO OUT

S7   NROOT=0
     GOTO OUT

S8   ROOT1=ROOT2
     NROOT=1
     GOTO OUT
```



Next, AMPIC transforms the FORTRAN type statements into mathematical-type statements (right side of next page). (Left side repeats previous results.)

The reason for this step is that the sequential nature of computer program statements hides the functional relationship between input and output. For example, the two FORTRAN statements of S2 (left side) translate into the single input-output relation shown (right side). With an appropriate symbolism to indicate output ("<NU>") and input (no extra symbol), the functional effect of each Sn can be stated in terms of equalities, inequalities, and the effect of each Tn as the satisfaction of logical conditions.

AMPIC FORTRAN EXAMPLE: TRANSLATION (PHASE III)

```
T1   IF(A.EQ.0.) GOTO S7

S2   TEMP1=4.*A*C
     TEMP1=B*B-TEMP1

T3   IF(TEMP1.LT.0.) GOTO S7

S4   TEMP2=2.*A
     ROOT2=-B/TEMP2

T5   IF(TEMP1.EQ.0.) GOTO S8

S6   TEMP2=FSQRT(TEMP1)/TEMP2
     ROOT1=ROOT2+TEMP2
     ROOT2=ROOT2-TEMP2
     NROOT=2
     GOTO OUT

S7   NROOT=0
     GOTO OUT

S8   ROOT1=ROOT2
     NROOT=1
     GOTO OUT
```

```
T1:  JUMP       (A = 0.)
     NO JUMP    (A ≠ 0.)

S2:  TEMP1<NU> = B * B - 4. * A * C

T3:  JUMP       (TEMP1 < 0.)
     NO JUMP    (TEMP1 ≥ 0.)

S4:  TEMP2<NU> = 2. * A
     ROOT2<NU> = - B / (2. * A)

T5:  JUMP       (TEMP1 = 0.)
     NO JUMP    (TEMP1 ≠ 0.)

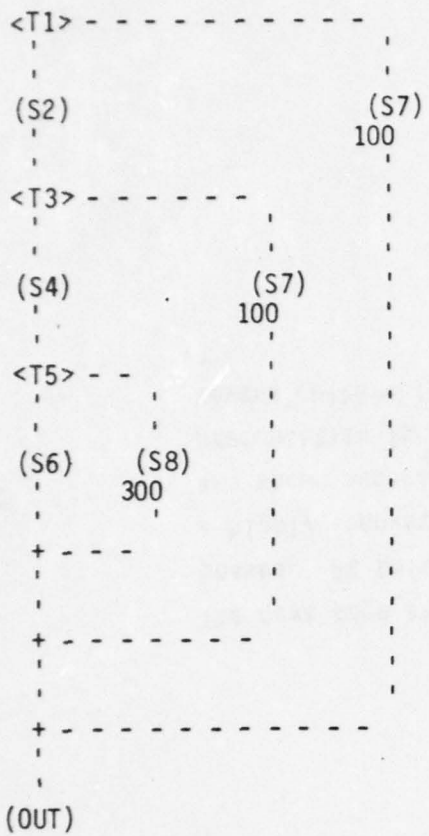
S6:  TEMP2<NU> = .FSQRT(TEMP1) / TEMP2
     ROOT1<NU> = ROOT2 + .FSQRT(TEMP1) / TEMP2
     ROOT2<NU> = ROOT2 - .FSQRT(TEMP1) / TEMP2
     NROOT<NU> = 2

S7:  NROOT<NU> = 0

S8:  ROOT1<NU> = ROOT2
     NROOT<NU> = 1
```

The next page shows side-by-side the AMPIC outputs for the structuring and translation phases. At this point, the user has all the information that would be provided by a highly sophisticated flowcharter, except that the contents of the flowchart boxes are shown separately from the flowchart diagram. The advantage of this method of presentation is the compactness of the diagram. For large diagrams, AMPIC provides several alternatives for reducing the total diagram into a set of sub-diagrams.

AMPIC FORTRAN EXAMPLE: FLOWCHART WITH TRANSLATIONS




```
T1:   JUMP          (A = 0.)
      NO JUMP       (A ≠ 0.)

S2:   TEMP1<NU> = B * B - 4. * A * C

T3:   JUMP          (TEMP1 < 0.)
      NO JUMP       (TEMP1 ≥ 0.)

S4:   TEMP2<NU> = 2. * A
      ROOT2<NU> = - B / (2. * A)

T5:   JUMP          (TEMP1 = 0.)
      NO JUMP       (TEMP1 ≠ 0.)

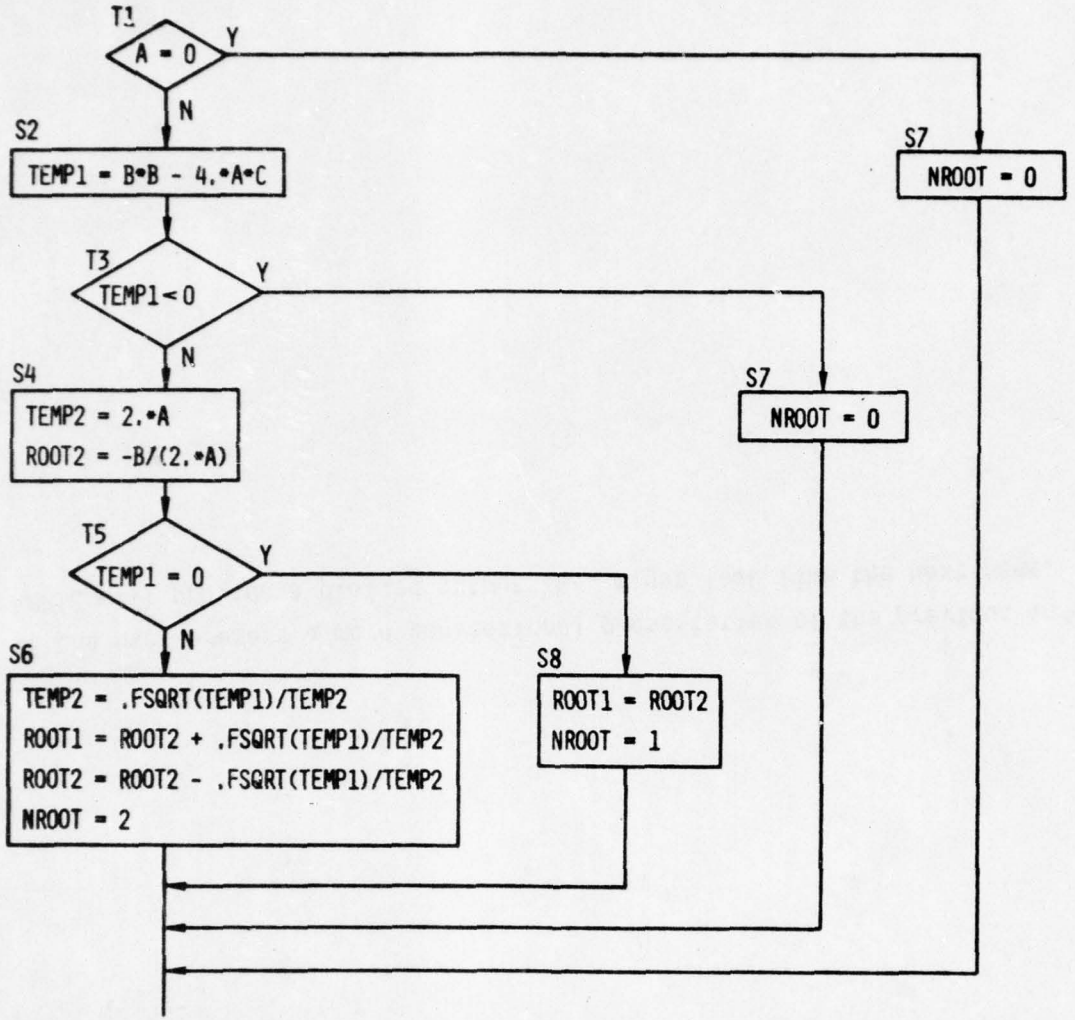
S6:   TEMP2<NU> = .FSQRT(TEMP1) / TEMP2
      ROOT1<NU> = ROOT2 + .FSQRT(TEMP1) / TEMP 2
      ROOT2<NU> = ROOT2 - .FSQRT(TEMP1) / TEMP 2
      NROOT<NU> = 2

S7:   NRROT<NU> = 0

S8:   ROOT1<NU> = ROOT2
      NROOT<NU> = 1
```

If the user prefers a more conventional presentation of the previous information, AMPIC will provide a plotted output that might look like the next page.

CONVENTIONAL FLOWCHART FORMAT



Next, AMPIC will use the previously derived information and provide input-output functional expressions for entire paths through the input module.

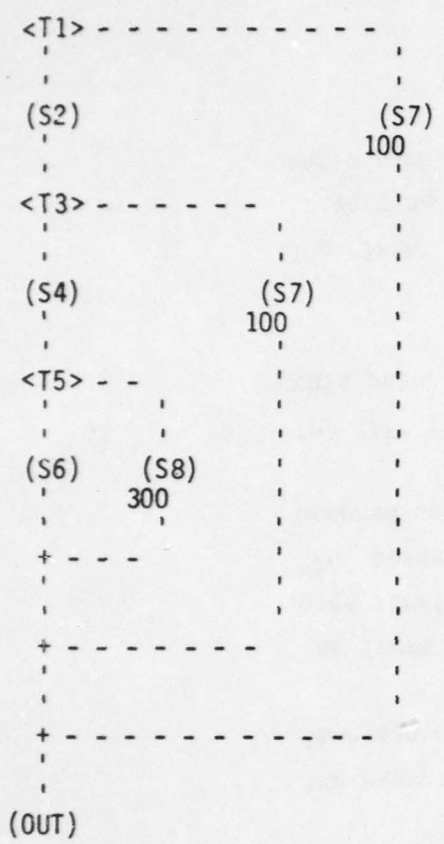
The examples on the next page show this for two of the four paths in the FORTRAN language module.

The first example (upper right) is the functional expression for the path that goes straight down from T1 to OUT. The line labeled "PATH" identifies this path. "Pn" means "go down" ("no jump") at Tn. Hence, (P1, S2, P3, S4, P5, S6, OUT) is an ordered set that describes the path.

The "IF" statement describes the three conditions to be satisfied in order to traverse this path. The conditions are with respect to the input values of A, B, and C.

The "THEN" statement describes the total input-output functional results of traversing this particular path. (The "<NU>" is now the output with respect to the whole path. Variables without "<NU>" are inputs with respect to the whole path).

AMPIC FORTRAN EXAMPLE: PATHWISE RESULTS (PHASE IV)



PATH: (P1,S2,P3,S4,P5,S6,OUT)

IF: (A ≠ 0.) & (B * B - 4. * A * C ≥ 0.) & (B * B - 4. * A * C ≠ 0.)

THEN:
```
TEMP1<NU> = B * B - 4. * A * C
TEMP2<NU> = .FSQRT(B * B - 4. * A * C) / (2. * A)
ROOT2<NU> = - B / (2. * A) - .FSQRT(B * B - 4. * A * C) / (2. * A)
ROOT1<NU> = - B / (2. * A) + .FSQRT(B * B - 4. * A * C) / (2. * A)
NROOT<NU> = 2
```

PATH: (P1,S2,P3,S4,R5,S8,OUT)

IF: (A ≠ 0.) & (B * B - 4. * A * C ≥ 0.) & (B * B - 4. * A * C = 0.)

THEN:
```
TEMP1<NU> = B * B - 4. * A * C
TEMP2<NU> = 2. * A
ROOT2<NU> = - B / (2. * A)
ROOT1<NU> = - B / (2. * A)
NROOT<NU> = 1
```

For our second example, we will use the assembly language program generated by a FORTRAN compiler which took as input the FORTRAN module of our first example.

The next page shows this assembly language module after it has gone through the first AMPIC phase.

AMPIC COMPILED CODE EXAMPLE: CODE PARTITIONING (PHASE I)

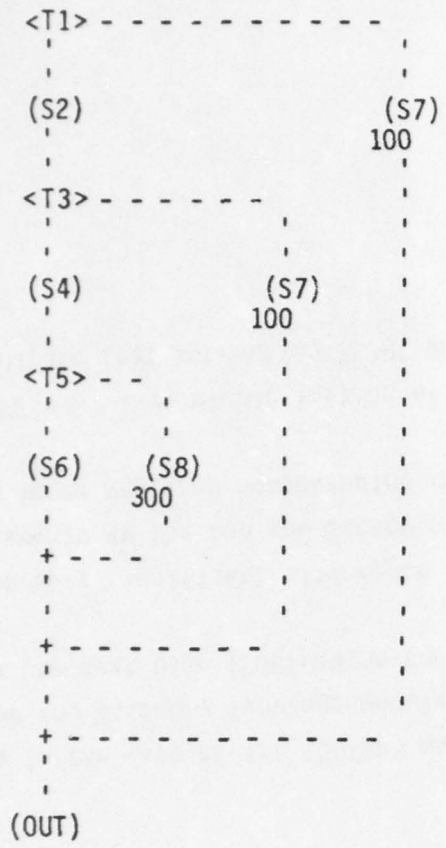
```
        OC1     OCT     203400
                OCT     000000
        ROOT    PZE     0
S1              SS      OLT1
                LX    1 OLT1
                SB    1 OLT2,1
                SB    1 OLT3,2
                SB    1 OLT4,3
                LX    0 NROOT
                DLA     A
T2              JUZ     100F
S3              DLA     OC1
                FMR     A
                FMR     C
                DSA     TEMP1
                DLA     B
                FMR     B
                FANR    TEMP1
                DSA     TEMP1
T4              JUN     100F
S5              DLA     A
                FAR     A
                DSA     TEMP2
                SLAL    36
                FANR    B
                FD      TEMP2
                DSA     ROOT2
                DLA     TEMP1
T6              JUZ     300F
S7              JAL     $FSQRT+1,$FSQRT
                PZE     TEMP1
S8              FD      TEMP2
                DSA     TEMP2
                DLA     ROOT2
                FAR     TEMP2
                DSA     ROOT1
                DLA     ROOT2
                FANR    TEMP2
                DSA     ROOT2
                LLK     2
                SL    0 NROOT
S9      9G      LB    1 OLT2,1
                LB    1 OLT3,2
                LB    1 OLT4,3
                LS      OLT1
                JI      1G
S10     100F    SLAL    36
                SL    0 NROOT
                J       9G
S11     300F    DLA     ROOT2
                DSA     ROOT1
                LLK     1
                SL    0 NROOT
                J       9G
```

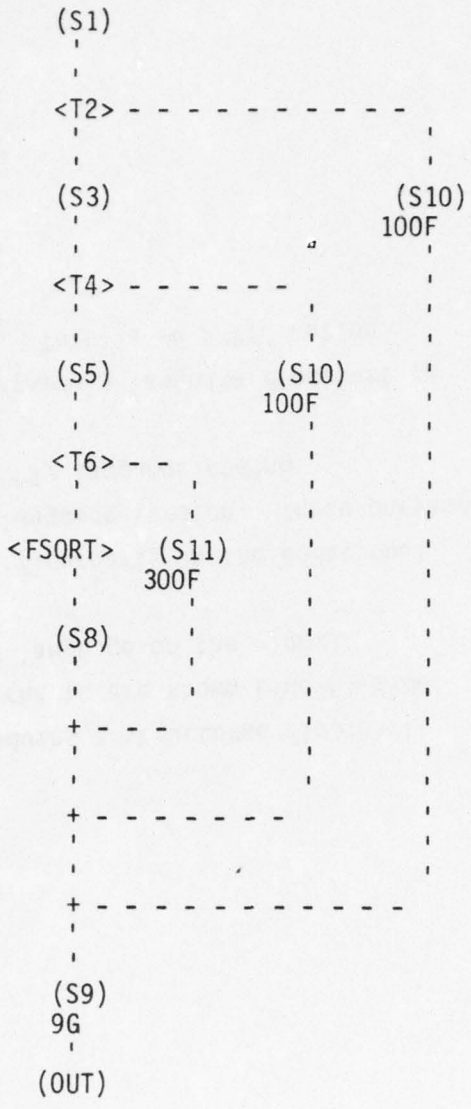
As in the case of the FORTRAN module, AMPIC produces a structured flowchart for the assembly language module. The two flowcharts are shown side-by-side on the next page (FORTRAN on the left, assembly language on the right).

The only significant difference between the two flowcharts is the additional elements at the top and bottom of the assembly language version. These correspond to index register housekeeping required in assembly language coding.

"<FSQRT>" was printed instead of "(S7)" in the assembly language flowchart to indicate that subroutine FSQRT was called at S7. This is an AMPIC option.

AMPIC COMPILED CODE EXAMPLE: STRUCTURED REPRESENTATION (PHASE II)





AMPIC translates assembly language code into the same mathematical-like language as in the FORTRAN case. The examples on the next page show how some of the code segments are translated.

The "$" notation distinguishes machine registers and compiler generated variables from program variables. E.g. $A is the accumulator.

AD-A033 625 ROME AIR DEVELOPMENT CENTER GRIFFISS AFB N Y F/G 9/2
PRESENTATIONS AT THE RADC/AR³A INVITATIONAL DOD/INDUSTRY CONFER--ETC(U)
AUG 76

UNCLASSIFIED NL

2 OF 5
AD
A033625



AMPIC COMPILED CODE EXAMPLE: TRANSLATION (PHASE III)

```
S3   DLA    0C1          S3:  $A<NU> = B * B - 4. * A * C
     FMR    A                 $Q<NU> = 0
     FMR    C                 $CD<NU> ≠ 1
     DSA    TEMP1             TEMP1<NU> = B * B - 4. * A * C
     DLA    B
     FMR    B
     FANR   TEMP1
     DSA    TEMP1

T4   JUN    100F         T4:  JUMP       ($CD = 1 & $LS) ∨ ($CD ≠ 1 & $AU < 0)
                              NO JUMP    ($CD = 1 & ~$LS) ∨ ($CD ≠ 1 & $AU ≥ 0)

S5   DLA    A            S5:  $A<NU> = TEMP1
     FAR    A                 $Q<NU> = .$QREM((- B) / (A + A))
     DSA    TEMP2             $CD<NU> ≠ 1
     SLAL   36                TEMP2<NU> = A + A
     FANR   B                 ROOT2<NU> = (- B) / (A + A)
     FD     TEMP2
     DSA    ROOT2
     DLA    TEMP1
```

AMPIC generages input-output functional expressions for assembly language modules just as it did for FORTRAN modules. The two cases shown on the next page are for corresponding paths in the two examples. (Top is FORTRAN, bottom is assembly language).

Except for the additional results relating to machine registers (the "$" cases), the results are equivalent, as they should be. Note, that the compiler produces "A + A" in place of "2. * A".

AMPIC COMPILED CODE EXAMPLE: PATHWISE STATEMENT (PHASE IV)

```
PATH:  (P1,S2,P3,S4,P5,S6,OUT)

IF:    (A ≠ 0.) & (B * B - 4. * A * C ≥ 0.) & (B * B - 4. * A * C ≠ 0.)

THEN:  TEMP1<NU> = B * B - 4. * A * C
       TEMP2<NU> = .FSQRT(B * B - 4. * A * C) / (2. * A)
       ROOT2<NU> = - B / (2. * A) - .FSQRT(B * B - 4. * A * C) / (2. * A)
       ROOT1<NU> = - B / (2. * A) + .FSQRT(B * B - 4. * A * C) / (2. * A)
       NROOT<NU> = 2


PATH:  (S1,P2,S3,P4,S5,P6,S7,S8,S9,OUT)

IF:    (A ≠ 0) & (B * B - 4. * A * C ≥ 0) & (B * B - 4. * A * C ≠ 0)

THEN:  $C<NU> = 0
       $AU<NU> = (- B) / (A + A) - (.FSQRT(B * B - 4. * A * C)) / (A + A)
       $AL<NU> = 2
       $XR0<NU> = 258 - .$INT1((258) / 256) * 256
       $XR1<NU> = .$INT1((258) / 256)
       $CD<NU> ≠ 1
       $OLT1<NU> = 258
       $OLT2<NU> = $B1
       $OLT3<NU> = $B2
       $OLT4<NU> = $B3
       TEMP1<NU> = B * B - 4. * A * C
       TEMP2<NU> = (.FSQRT(B * B - 4. * A * C)) / (A + A)
       ROOT2<NU> = (- B) / (A + A) - (.FSQRT(B * B - 4. * A * C)) / (A + A)
       ROOT1<NU> = (- B) / (A + A) + (.FSQRT(B * B - 4. * A * C)) / (A + A)
       NROOT<NU> = 2
```

We have shown through actual examples how AMPIC analyzes higher-order language or assembly language programs. Since AMPIC transforms both types into common mathematical-like expressions, AMPIC provides a way to verify what the compiler has done. AMPIC output can also be used as program documentation. As demonstrated, AMPIC can be used to understand what a complicated program operates.

Verification of Requirements and Design Through Structured Analysis Techniques

Raymond J. Rubey
SOFTECH

Software verification and validation activities have as a primary objective the discovery and correction of errors in the computer programs under examination. Many of these errors are introduced by the programmers during the programming activity and go undetected during the subsequent testing activities. However, a significant number of errors are introduced earlier in the software development process even before the programming effort. An examination of a dozen different aerospace verification and validation activities revealed out of a total of 649 significant errors that were discovered, 246 errors had their origins in the software specifications. Another study indicated that in command and control software, as many as two-thirds of the errors discovered during and after acceptance testing had their origins in the software specifications.[1] If one examines the literature one discovers that many new techniques and tools have been produced that, through improvements in the programming and testing activities, increase software reliability.[2,3,4,5] There has not been an equivalent effort devoted to improving the techniques for the detection or elimination of errors due to inadequacies in the software specifications. Does this mean that our existing techniques for the review of software specifications are adequate? In the examination of the dozen aerospace verification and validation activities cited above, 145 of the 246 errors in the software specifications were detected by a thorough review of those specifications; 101 errors went undetected by that review, were passed on to the coding and test activity, and were only detected during the validation of the resultant code. The failure to detect errors in the software specifications early in the development process can lead to increased software development cost. It can cost 10 to 60 times more to correct an error found during testing than it would have cost if that error had been detected before programming was begun. Another factor is that the organization of software testing often means that the most significant errors are detected late in the testing activity. The initial unit testing usually only detects the errors made by the programmers when the unit under test was programmed. The integration testing usually only detects errors that result from an incomplete or ambiguous software design. Acceptance testing, done last, detects errors that were made very early, when the functions that the program was to do were defined. Clearly the early verification of the original functional requirements is essential. Equally clear is the fact that the repertoire of techniques that has been used in the past to verify software specifications and functional requirements needs to be enhanced.

One such enhancement is the use of Structured Analysis for verification. Structured Analysis was developed and is currently used to develop functional requirements ab initio.[6,7] The use of Structured Analysis will

improve software reliability because of the increased completeness and correctness of the functional requirements just as the use of structured programming improves software quality by reducing the complexity of the resultant program. However, there is the need to verify the completeness and correctness of software functional requirements that were not developed through Structured Analysis methods. Structured Analysis can contribute to satisfying this need. Before describing how this can be accomplished a short discussion of the basic concepts of Structured Analysis is in order.

Structured Analysis is a method for modeling complex systems in a top-down hierarchic form. The model that results from the application of Structured Analysis consists of an ordered sequence of "blueprint" diagrams. Each diagram makes a complete statement to a clearly understandable level of detail about a specific topic. Further detail about particular functions is presented on lower level diagrams. Each lower level, more detailed, diagram connects exactly into the higher level, more general diagram in a manner that preserves the logical relationship of each component to the total system.

A complete model consists of two types of decomposition:

1) An activity model that indicates the interrelationships between the activities the system is to perform in terms of the input data, output data, control data and mechanisms that constrain each activity.

2) A data model that indicates the inter relationships between data in terms of the generating, using and controling activities and the storage mechanisms that constrain the data. The completeness and consistency of the two models is demonstrated by an activity-data tie which links each element of one model with its equivalent in the other model.

The graphical language used to construct Structured Analysis models is very simple consisting of boxes to represent activities (in the activity model) or data (in the data model) and arrows to represent the constraints. A diagram, at any level, can have no fewer than three and no more than six boxes. This prevents any diagram from becoming overly complex and forces decomposition to show details. Input arrows are always drawn entering the left side of a box, controls entering the upper side, outputs leaving the right side and mechanisms entering the bottom side. A referencing scheme is used in Structured Analysis to assign unique and meaningful numbers to each activity or data box to enable rapid cross-referencing between levels. Finally a key element of Structured Analysis is a disciplined approach to the process by which the diagrams are produced and reviewed.

In developing a functional requirements model from the beginning using Structured Analysis, the responsible analysts would attempt to capture the experiences of experts and users, the intent of the needs analysis documentation, and the influences of the environment in which the software system must be developed and operate. In verifying an existing functional requirements document, which has not been developed using Structured Analysis techniques, the responsible analysts attempt to capture the content of that existing functional requirement document in a Structured Analysis model. Errors in the functional requirements document are revealed in two ways. First, attempting to restate the requirements in the more rigid and structured format will reveal inconsistencies, missing or incomplete interfaces, and statements that are either too general or too specific. Although simple in appearance, the Structured Analysis language can be a tough task-master; poorly-conceived and poorly-expressed elements of the software requirements documents are recognized as such when one attempts to construct the model. Secondly, the Structured Analysis model is more accessible for review and comment than the original requirements documentation because of the models hierarchial form and explicit description of interfaces. The simplicity of the Structured Analysis is a great advantage in this respect. Individuals with little or no software knowledge can be taught how to "read" the diagrams of the model in less than two hours. This enables the intended users of the software, who might be deterred from reading the original requirements documents because of their bulk and unfamiliar language, to participate in an effective review of the functional requirements.

One of the first things that is observed when one attempts to construct a Structured Analysis requirements model from the typical requirements documentation is that that original documentation is a mixture of incompatible levels of detail. Some requirements will be stated very generally without there being any further decomposition or detailing. Other requirements will be stated in a very specific and detailed manner but the general requirement that provides the motivation for these detailed requirements is not provided. If one were to imagine the requirements as being organized as a tree structure, proceeding from the general to the specific, one would observe that many requirements documents have many fine and detailed branches unattached to the main limbs and many large limbs that possess no smaller, more detailed branches. The missing portions of a functional requirements document will become apparent as the model is constructed.

The use of Structured Analysis is of benefit in initially defining a requirements/design/code traceability matrix. The normal requirements documents are relatively unstructured from a requirements viewpoint. The same requirement may be repeated with different emphasis and from a different viewpoint at several places in the documents. Software requirements may not be clearly differentiated from hardware requirements. Detailed, specific requirements are interspersed with general, overall requirements. The construction of a requirements model will partition general and specific requirements, will show the relationship of software requirements to hardware requirements, and will point out duplications or ambiguities. Each

requirement of the model will have a unique "node" number that is correlated both to higher and lower requirements in the hierarchy and to the original requirements documents. This node number can then serve as an identifier in matching the design or code, which themselves will be more naturally structured than the original requirements, to the requirements.

The use of Structured Analysis to verify specifications is a relatively recent application of the technique. Further experience will provide greater knowledge about the types of errors it can and cannot find. In applying this technique for verification, one must constantly bear in mind that one is not constructing alternate or replacement functional requirements documents. Rather one is attempting to restate, as accurately as possible, what is in the original functional requirements documents. This can be difficult and the degree of difficulty is proportional to the lack of structure and completeness of the original documents. Inferring the structure that may be inherent in a mass of detail requires an understanding of that detail.

In Structured Analysis one should focus first on what functions should be accomplished by the software. Only later, when these functions are defined should one annotate the model to include timing requirements. Unfortunately, many real-time systems have their timing considerations almost inextricably mixed with actual functional requirements. For example, a requirement specification may state that functions A, B and C are to be performed every second, and functions D, E and F every minute. However, the grouping of functions A, B and C may only be a consequence of the fact that they need to be performed at a higher rate than functions D, E and F. In the model the functions to be performed must be separated from such timing requirements so that the model's structure is dictated by functional considerations rather than by the less binding timing considerations. Similarly, the model must be constructed with the functional requirements separated from the implementation dependent considerations.

Even after a model is constructed, there may be some difficulty in communicating to the original requirement specification's developer the deficiencies in the specification. As was indicated earlier, the model is not a replacement for the original specification, and as a consequence all criticisms of that specification should be expressed in a way that is independent of the model. It may be argued that, if the model is, or can be made to be a superior definition of the functional requirements, the model should be used by the developer as a basis for his subsequent development activities. This, however, is undesirable, as it would destroy the separation of roles between the developer and validator.

The work that has been accomplished does indicate that Structured Analysis will be beneficial in the review of specifications as well as in the writing of specifications. Structured Analysis does provide an organized method and discipline for the modeling of specifications that is compatible with the concepts of software engineering. The use of Structured Analysis can lead to the early detection of deficiencies in software specifications.

REFERENCES

1. Some Experience with Automated Aids to the Design of Large-Scale Reliable Software
B. W. Boehm, R. K. McClean, D.B. Urfrig, TRW,
Proceedings of the 1975 International Conference on Reliable Software, IEEE, pgs. 105-113, April, 1975

2. Automated Aids for Reliable Software, Donald J. Reifer,
The Aerospace Corporation, Proceedings of the 1975 International Conference on Reliable Software, IEEE/ACM, pgs. 131-142, April 1975

3. A View of Program Verification, Ralph L. London, USC Information Sciences Institute, Proceedings of the 1975 International Conference on Reliable Software, IEEE/ACM, pgs. 534-545, April 1975

4. Approaches to Improved Program Validation Through Programming Language Design, Donald K. Kosy, Rand Corporation, pgs. 75-97 of Program Test Methods, Prentice-Hall, 1973

5. Top-Down Structured Programming Techniques, C. L. McGowan and J. R. Kelly, Petrocelli/Charter, New York 1975

6. Structured Analysis for Requirements Definition, D. T. Ross and K. E. Schoman, Jr. Second International Conference on Software Engineering , San Francisco, October 1976 (To Be Published)

7. An Introduction to SADT (TM) SofTech, Inc., Document No. 9022-78, February 1976

NEW DIRECTIONS IN AUTOMATED TOOLS FOR IMPROVING SOFTWARE QUALITY

BY

LEON G. STUCKI
COMPUTER SCIENCE DEPARTMENT
MCDONNELL DOUGLAS CORPORATION
HUNTINGTON BEACH, CALIFORNIA
TELEPHONE: (714) 896-3774

to appear in forthcoming book:

Yeh, R.T., et.al., Current Trends in Programming Methodology, Vol. 2, Program Validation, Prentice Hall, to be published early 1977 (also available from author).

The Production of Quality Software Is A Problem

At the time Jules Verne was writing his fictional novel on a journey to the moon, no one took him seriously. But yet in the early 1960's as President J. F. Kennedy issued his goal of reaching the moon within a decade the task appeared to be much more reasonable. The major difference between the 19th and 20th century attitude can be attributed to the new tools like the computer which have greatly extended man's scope of feasible problems.

As with many of man's tools one must utilize the computer with some degree of caution.

Programming has been described by Dijkstra as the most complex mental activity ever undertaken by mankind[1]. Advances in computer hardware capabilities have brought previously unsolvable problems into the scope of solvability. Man's rapidly increasing reliance upon the correct behavior of complex software systems is of definite concern. As new systems realizing more complex hardware and software functions continue to evolve, accurate and efficient implementation of these systems are critical considerations.

BACKGROUND AND SIGNIFICANCE

The concept of "structured programming" associated with Dahl-Dijkstra-Hoare,[2] Mills[3], et.al., was originally suggested as a vehicle for improving software quality through a disciplined development of algorithms. The use of a restricted set of control structures within programs was one part of that concept. Today, common use of the term "structured programming" is associated with so many diverse activities that Dijkstra, the term's originator, now attempts with great vigor to avoid its use.[4]

Wirth presents another very interesting treatment of the processes for methodically and systematically designing algorithms. Wirth goes beyond the notions of restricted flow of control and addresses issues of data representation and operations allowed on data in his development of PASCAL.[5]

In an attempt to bring the benefits of "structured programming" to the common people many zealous prophets have arisen each with their own new set of restricted control structures and their associated bag of tools, usually a preprocessor, for allowing the use of these control structures with existing languages (e.g. FORTRAN and COBOL)[6].

Another set of tools has also been introduced over the last few years which deals with control flow through programs.[7-13] Software probes or instrumentation are automatically placed into a program for monitoring the dynamic execution behavior of an algorithm. Software probes in the form of source language statements are inserted into the source code to gather statistics during program execution. These probes can provide insight into many aspects of algorithmic behavior beyond a simple flow of control analysis. The notion of building self-metric (self-measuring) software has been introduced previously by this author[14], however, significant expansion of this concept is now being explored as a vehicle for improving software quality[15-21].

In order to illustrate the type of automated tool capabilities currently available and some of the new techniques now under consideration, the tool most familiar to the author will be described. It is hoped that this currently operational system will offer some insight into the concept of self-metric software and show a few of the measurement schemes available for dynamic program analysis.

AUTOMATED TOOLS

The Program Evaluator and Tester System - A Program for Generating Self-Metric Software

Ongoing research is being conducted by McDonnell Douglas Astronautics Company in the area of software validation. Several powerful tools have been and are currently being developed for this purpose. A FORTRAN implementation of the Program Evaluator and Tester (PET) demonstrates the value of a self-metric testing approach for higher-level languages. Similar tools for other source languages are also being developed. [13,14,20]

These tools essentially "instrument" the software under test by inserting the software equivalent of sensors into the subject programs. This has the effect of making the programs self-measuring. Each time the software under test performs a significant event, the occurrence of this event is recorded, with the exact nature of the recording process depending on the measurements desired and the type of event performed.

Source Program Instrumentation

Figure 1 shows a sample FORTRAN program segment and its related flow diagram. Optional instrumentation points are identified for three sets of sensors applicable to this particular program segment. The execution-count sensors are shown at all points on the flow diagram where the internodal traversal frequency is not obvious.

Copy available to DDC does not permit fully legible reproduction

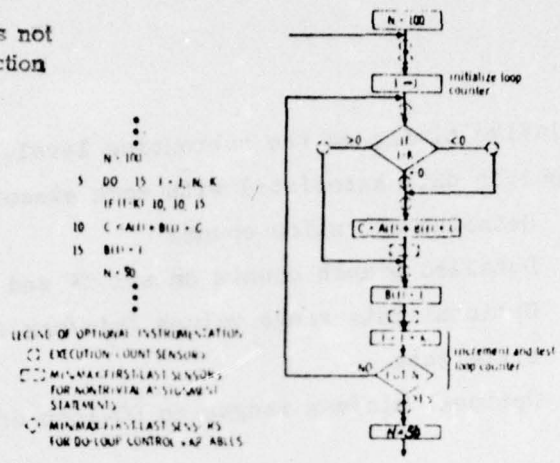


Figure 1. FORTRAN Program Segment Showing Optional Instrumentation

These execution-count sensors can be most efficiently realized by associating a unique counting cell with each sensor and inserting these sensors only at those points where logical program breaks occur. The min/max/first/last sensors for assignment statements appear following non-trivial assignment statements. A trivial assignment statement, or one in which a constant or literal is assigned to a variable, causes no instrumentation to be generated. The min/max/first/last sensors for DO-loop control variables are inserted after the label or statement number, if present, and before the invocation of the loop if variable parameters are used in any of the three loop-controlling fields.

Two techniques have been experimentally used in implementing these and several other types of software sensors: (1) direct code insertion, and (2) invocation of run-time routines. The direct code insertion appears to be faster in most cases but the run-time routine is more flexible in that measurements can be more easily altered at execution time. The current PET system utilizes a hybrid scheme employing the selective use of both types of instrumentation.

As a result of running the instrumented program, a profile is produced containing part or all of the following measurements:

1. The number and percentage of all potential executable source statements which were executed one or more times.
2. The number and percentage of those program branches taken.
3. The number and percentage of those subroutine calls which were executed.
4. The number of times each subroutine was called, together with a list of those subroutines that were never entered.

5. Relative timing on the subroutine level.
6. Specific data associated with each executable source statement.
   a. Detailed execution counts
   b. Detailed branch counts on all IF and GOTO statements
   c. Optional data range values (min/max/first/last) on assignment statements.
   d. Optional min/max ranges on DO-loop control variables.

These summaries and detailed reports can be employed to establish a figure for the degree of testing to which the program structure has been subjected.

The measurements performed allow us to examine the internal structure of a software system rather than merely treating it as if it were a "black box". While the testing process does not technically prove the correctness of the algorithms used, it does allow us to observe the behavior of the algorithms with actual test data. From these observations, the effectiveness of the software validation process can be greatly improved.

Appendix A contains a number of sample reports produced by the PET system along with accompanying descriptions explaining their use.

Automated Program Support System Concept

The concept of an Automated Program Support System offering assistance in several new areas of the software engineering process is currently being formulated to incorporate present tools and techniques within a comprehensive collection of automated aids for software development and certification. This extensible system contains five major components:

1. A Program Development Subsystem for guiding the development of well structured programs in a top-down fashion;
2. A Test Statistics Subsystem for both gathering static measurements and monitoring the dynamic behavior of software (an extension of MDC's currently existing PET system) (see Appendix A);

3. A Documentation Subsystem for assisting in the generation and maintenance of detailed software documentation;
4. A Simulation Subsystem for monitoring the behavior of assembly language software and scaling real-time data used for validation testing; and
5. A Test Data Generation Subsystem for synthesizing and assessing tests of critical software components.[22]

Each component tool of this system may be applied to a given program separately, although the tools can be most effectively used in a disciplined rational sequence.

Current Areas of Research

The balance of this paper will be restricted to remarks describing new research currently underway on the Test Statistics Subsystem referenced above.

There is currently a large gap between exhaustive testing of all potential program paths and the statement that all program branches or segments have been exercised at least once. Advance research efforts are being conducted in several areas in an attempt to help bridge this gap.

One very promising area involves the incorporation of an assertion capability into existing languages.

Assertion and Monitor Capabilities for Existing Languages

A number of new languages are currently being investigated by researchers for possible use in proving the correctness or consistency of programs. Although the various proof techniques differ to some extent, one common requirement is the development of assertions describing the nature of the algorithms. The program provers are currently exploring techniques for checking the consistency of these assertions in a static logical sense for specialized languages.

The assertion concepts for programming languages which are now being developed constitute major extensions to our ability to carry out "systematic programming". These new assertion concepts impact all phases of the software life cycle from initial requirements and design phases down through certification, and maintenance iterations. These assertion concepts are designed to encourage the development of algorithmic validation criteria as the implementation evolves from the initial algorithm requirements and specifications down to the final program code. The idea can be illustrated by Figures 2 and 3 which show how the point at which validation and/or other quality criteria can be attached directly to the software definitely impacts the quality of the end software and the effort needed to validate and maintain that software.

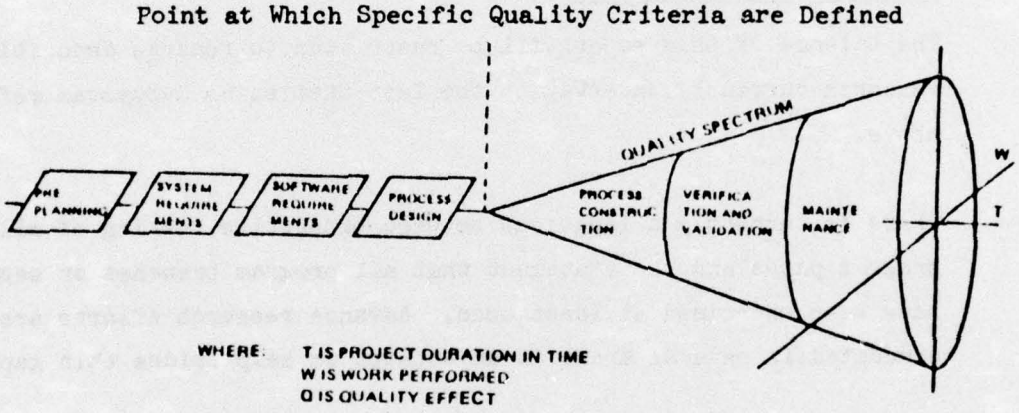


Figure 2 Effect of Delaying Quality Considerations

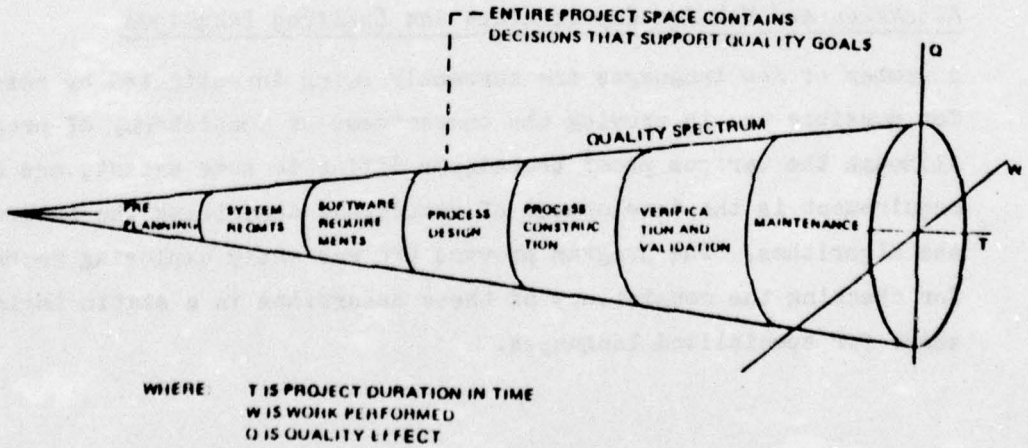


Figure 3 Effect of Continuous Quality Consideration

ASSERTION CONCEPTS FOR IMPROVING SOFTWARE QUALITY

Generalized Local Assertion

A generalized local assertion may be embedded in a comment at any point within the executable code of a program where another executable statement may appear. The local assertion is designed to enhance the documentation of critical algorithms throughout the entire life cycle of the software. Dynamic execution time checks can be activated at selected points in time to ensure that the actual run-time environment is consistent with the logical state specified in the assertion. This dynamic assertion checking can be used to great benefit in debugging, validation, certification, and maintenance of complex systems.

The format of the generalized local assertion is:

ASSERT LOCAL(extended-logical-expression)[optional-qualifiers]
[control-options]

The exact placement and treatment of the assertions will be tailored to the existing language facilities in currently defined languages. In these currently available languages the assertions will be implemented through specialized comments processed by a source code preprocessor. New language development and future compilers for existing languages may contain options for directly implementing the assertions.

Optional-Qualifiers

In order to provide an existential and universal qualifier notion to the generalized local assertion an optional looping capability is defined:

...FOR $\left[\begin{matrix}\text{ALL}\\ \text{SOME}\end{matrix}\right]_{1}^{n}$ [(variable-list) (set of ranges/values)]

WHERE (qualifier-controlling-logical-expr)

e.g. /* ASSERT (X(I) ¬=X(J)) FOR ALL (I,J) (1:8) WHERE (I ¬=J)*/

means: [$\forall$I,J such that $1 \leq I$, $J \leq 8 \wedge I \neq J$] $\rightarrow$ $X(I) \neq X(J)$

Assertion Control Options

The total control alluded to above (i.e., ignoring all assertions by treating them as comments) offers the user a binary choice as to whether or not to apply dynamic assertions during program development; however, other levels of control are provided within the assertion language itself.

The assertion language itself contains three hierarchical levels of control:

1) instrumentation control - control of those sets of assertions which will be instrumented at a given level of testing,
2) dynamic control - run-time control of those instrumented assertions which are to be dynamically checked, and
3) threshold control - user control when assertion violations are observed.

Instrumentation control is provided by a LEVEL option. The syntax of this option is:

...LEVEL (preprocessor-control-expression)....

The LEVEL option provides information to the preprocessor telling it which sets of assertions should be considered for dynamic analysis. This level of control provides a means for testing selected software features at various points within the software development cycle and fits in well with the top down approach to program development. This also allows a user to group sets of assertions together for various types of dynamic checks.

Dynamic control is provided by a CONDITION option. The syntax of this option is:

...CONDITION (dynamic-control-expression)...

The CONDITION option provides run-time control of the assertions which have been built into the program. This option effects only those assertions which have been actually instrumented, thus the CONDITION option is of lower priority than the LEVEL option. It should be also noted that the CONDITION option can be dynamically changed under program control to activate or de-activate the assertion as often as desired.

Threshold control is provided by a LIMIT option. The syntax of this option is:

...LIMIT n[VIOLATIONS] [|HALT / EXIT [VIA] proc-name|] ...

The LIMIT option provides user control in the event of n violations of the corresponding assertion. The user can specify that control be transferred to a wrap-up procedure proc-name if the EXIT phrase is specified. Otherwise, the HALT phrase will simply terminate execution and generate an assertion report automatically if n assertion violations are encountered. Motivated by a need to make assertions about arrays as well as scalars, the following notation has been adopted.

Array Notation for Assertions

Two areas of concern immediately arise when discussing data arrays, namely, array indices and array values. Thus, if one is monitoring program behavior, it is not enough to monitor array values alone, since program logic is invariably concerned with where these values are stored within the array.

The approach is to generalize the assertion and monitor capabilities to include data arrays. Array notation is as follows:

> Assume an array of the form $A(I_1,I_2,I_3,\ldots I_n)$. References to specific subsets of array values or array indices are indicated by $A(I_1', I_2', I_3' \ldots I_n')$, where I_i' is a subrange of I_i. This notation is position dependent; i.e., if I_2' is not referenced, its position must be indicated by an asterisk (*), as in $A(I_1',*,I_3' \ldots I_n')$. The format of each I_i is $\ell:\mu$ where $\ell<\mu<I_i$. if $\ell=\mu$ then the pair $\ell:\mu$ may be replaced by the simple token μ, as $A(\ell_1:\mu_1,\mu_2,\ldots)$. Thus, for A(10,20) we might reference
>
> A(5,10:15)
> A(*,3)
> A(2:5,*)
> A(2:6,2:10)

Extended Logical Expressions

Two types of extended local operations have initially been defined for the assertion language. An array to scalar logical operation will be allowed with its result being defined as 'true' if and only if all component to scalar operations are found to be true. An array to array logical operation will be allowed for identically specified array cross-sections with its result being defined to be true if and only if each pairwise component operation yields a result of 'true'.

Local Assertion Examples

A simple local assertion example is shown below in a typical report format. The assertion simply indicates that at the point where it is inserted into the source code we expect the value of the variable MOVE to be less than 9. The report format indicates that this assertion was checked 9 times. Violations were noted on the 6th and 7th executions of the assertion. It is furthermore noted that MOVE actually contained the value 9 on those two instances. A snapshot is taken of all pertinent variable values associated with the violation when the trace mode is specified.

```
                                          EXECUTION
                                           COUNT        SPECIFIC EXECUTION DATA

                                                  ASSERTION VIOLATIONS      2
C   ASSERT LOCAL (MOVE .LT. 9) LIMIT 10   9      EXEC NUMBER   VARIABLE VALUE
                                                       6          MOVE      9
                                                       7          MOVE      9
```

It is also worth noting that had we encountered 10 violations we would have halted execution at this point in the program.

Examples of the use of array cross-sections in extended logical expressions include the following: (assume an array A(10,20) has been defined)

(a) ASSERT LOCAL (A(*,3) .LT. 10) LIMIT 6 VIOLATIONS
(b) ASSERT LOCAL (A(2:6,2:10) .NE. 0)
(c) ASSERT LOCAL (A(*,*) .GT. 0)

In (a), the value of each array element whose second subscript=3 is checked and reported as a violation if its value is not less than 10. Ten array values will be checked in all. Any number of assertion violations within an array operation cause the operation to be counted as a single assertion violation. Thus, the HALT ON 6 parameter is concerned with only the number of invalid operations not with the number of violations within the array.

In (b), only array values within the specified subscript ranges are checked for an assertion violation. In (c) all array values are checked for an assertion violation.

Specialized Local Assertions

A number of additional specialized local assertions are proposed to facilitate the expression of user validation criteria. This extensible attribute of the local assertion concept is illustrated by the following constructs:

```
ASSERT LOCAL VALUE[S] (variable-list) (set-of-legal-ranges-and/or-values)...
ASSERT LOCAL VALUE[S] (variable-list) NOT (set-of-illegal-ranges-and/or-values)...
ASSERT LOCAL VALUE[S] (variable-list) INVARIANT...
ASSERT LOCAL SUBSCRIPT RANGE (list-of-array-specifications)...
ASSERT LOCAL ORDER (array-cross-section) ||ASCENDING|| ...
                                        ||DESCENDING||
```

All of these specialized local assertions could be replaced by one or more generalized local assertions, however, their existence facilitates the graceful transition from program requirements and their associated validation criteria to embedded program documentation in the evolving code.

These constructs cause instrumentation to be generated at the position where they occur or at the next executable statement. Forms three and four will monitor the execution of the following statements to insure that it does not alter the value of an invariant variable (e.g., through side effects from subroutine or function calls) or use subscripts outside the specified ranges.

The ASSERT ORDER statement checks a sequence of array values as follows:

```
ASSERT LOCAL ORDER (A(*,3)) ASCENDING
```

For an array A(10,20), the following assertion violation summary illustrates the type of information traced for a violation:

```
EXECUTION
COUNT                          SPECIFIC EXECUTION DATA
229              ASSERTION VIOLATIONS                1
                   EXEC NUMBER  SEQUENCE SNAPSHOT VALUE
                        18          A(7,3)         6
                                    A(8,3)         100*
                                    A(9,3)         8
```

The ASSERT VALUES statement checks variable values against a specific set of ranges and/or values to assure that it lies within the desired range. Distinct ranges are specified by pairs "min-expr:max-expr". Each range pair or distinct value within a set of value specifications is separated by a comma. For example, the following assertion could be used to check the value of the variables x and y.

```
/* ASSERT LOCAL VALUES (X,Y) (0,3,5:7,11:13,21)*/
```

In the example the current values of X and Y would be compared with the set:

```
(0,3,5,6,7,11,12,13,21)
```

Any discrepancies (i.e. any current values not in the set of specified legal values) would then constitute an assertion violation. The ASSERT SUBSCRIPT RANGE statement will check addressing on those arrays specified to ensure that only those portions of the array specifically selected are accessed. A subsequent example will illustrate the usefulness of this concept later in this paper. All of these latter constructions will result in providing similar traces to those already presented for out of bound conditions.

A TRACE statement is available to allow the user to control the number of execution snapshots reported for local assertion violations.

The format of the TRACE statement is:

```
TRACE  [ [ FIRST ] ]      n      [VIOLATIONS]
       [ [ LAST  ] ]
       [ [ OFF   ] ]
```

If a TRACE statement is not coded in the source program, detailed traces for local assertion violations will not be reported. If TRACE statements are coded, the first TRACE statement encountered causes the first (or last) n violations of subsequent local assertions to include snapshot information. Any subsequent TRACE statements encountered reset the values specified by the previous TRACE statement. TRACE OFF halts the reporting of execution snapshots for local assertions until the next TRACE statement is encountered.

The Concept of a Global Assertion

Expanding our notion of assertions, we immediately identify the need to expand to scope of application for our asserted program properties. In an effort to avoid requiring several similar local assertions within a particular program region, the concept of a global assertion has been introduced. This is a novel approach which promises to have a significant impact on the way we design, implement, and test software.

Global assertions will allow us to extend our capacity to inspect certain behavioral patterns for entire program modules, selected regions of modules, or module interfaces (entries and/or exits). Global assertions appear in the declaration section of the program module.

Formats include:

```
ASSERT  [ [ GLOBAL                    ] ]    (extended-logical-expression)...
        [ [ REGIONAL(region name)     ] ]
        [ [ ENTRY                     ] ]
        [ [ EXIT                      ] ]
```

```
                  [ { GLOBAL                          } ]
ASSERT            [ { REGIONAL (region-name)          } ]    VALUES...
                  [ { ENTRY                           } ]
                  [ { EXIT                            } ]

                  [ { GLOBAL                          } ]
ASSERT            [ { REGIONAL (region-name)          } ]    SUBSCRIPT RANGE...
                  [ {                                 } ]

                    { ENTRY                           }
ASSERT              { EXIT                            }      ORDER...
                    {                                 }
```

These global assertions will have effect within the scope defined (i.e., globally at all pertinent points, regionally over the named region, collectively for all entries and/or all exits.)

The VALUES statement inspects each specified variable as its value changes and reports when: (option 1) the new value is not one of the specified legal ranges and/or values, or (option 2) the new value assumes a specified illegal range and/or value, or (option 3) checks to make sure the values of the selected variables are preserved (i.e., no direct or externally caused changes are permitted).

The ASSERT SUBSCRIPT RANGE statement verifies that array subscripts fall within a specified range whenever the array is referenced during program execution. It should be noted that this statement provides a means for checking portions of arrays as well as normal upper and lower bounds. For this reason, it is more powerful than the PL/I type ON SUBSCRIPT RANGE check.

Instrumentation will be inserted into the source program by the preprocessor to accumulate the following statistics relative to assertion violations:

(1) Identify the statement that caused the assertion violation. For that statement an execution count and violation execution counts identical to those obtained for local assertions are reported.

(2) The actual value that caused the violation. This value is linked to the statistics identified in (1) above.

A GLOBAL TRACE statement will be available to allow the user to control the number of execution counts and associated data values reported for global assertion violations. The format of this statement is:

GLOBAL TRACE [FIRST / LAST / OFF] n [VIOLATIONS]

Some FORTRAN examples follow:

```
                .
                .
                .
20              DIMENSION A(10,20)
21      C       GLOBAL TRACE 10 VIOLATIONS
22      C       ASSERT VALUES (I,J,K,L) (0:100)
23      C       ASSERT VALUES ((II,LL) (-10:10)
24      C       ASSERT VALUES (KK,NN) (2,4,6,8,10)
25      C       ASSERT SUBSCRIPT RANGE (A(*,3))
26      C       ASSERT VALUES (X,Y,Z) INVARIANT

                .
                .
                .
102             K = K + 1
103             II = A(L,J) + LL
                ..
                .
                .
234             K = A(J,K) + I*100
235             II = II + 2
236             NN = KK*(I-J)
                .
                .
                .
300             CALL ROUTINEX(X,Y)
                .
                .
                .
```

If assertion violations occurred in this example, the following statistics are indicative of what would be reported by the Postprocessor:

	Annotated Program Listing	Execution Count	Specific Execution Data
102	K = K + 1	511	ASSERTION VIOLATIONS 1 ASSERT VALUE (K) (0:100) EXEC NUMBER VALUE 10 101
103	II = A(L,J) + LL . . .	511	ASSERTION VIOLATIONS 3 ASSERT VALUE (II) (-10:10) EXEC NUMBER VALUE 22 20 ASSERT SUBSCRIPT RANGE (A(*,3)) EXEC NUMBER VALUE 5 A(12,3) 105 A(1,4)
234	K = A(J,K) + I*100	125	ASSERTION VIOLATIONS 4 ASSERT VALUE (K) (0:100) EXEC NUMBER VALUE 52 101 53 102 ASSERT SUBSCRIPT RANGE (A(*,3)) EXEC NUMBER VALUE 52 A(5,4) 53 A(6,4)
235	II = II + 2	125	ASSERTION VIOLATIONS 1 ASSERT VALUE (II) (-10:10) EXEC NUMBER VALUE 50 12
236	NN = KK*(I-J) . . .	38	ASSERTION VIOLATIONS 1 ASSERT VALUE (NN) (2,4,6,8,10) EXEC NUMBER VALUE 20 7
300	CALL ROUTINEX(X,Y) . . .	53	ASSERTION VIOLATIONS 1 ASSERT VALUE (X) INVARIANT VALUE OF CALL PARM X EXEC NUMBER BEFORE CALL AFTER CALL 30 -10 -20

SUGGESTED USE OF ASSERTIONS

Basic Philosophy

Assertions should be developed and incorporated into a program during program design and not after-the-fact.

A Suggested Approach

1) Prior to generating any detailed code.
   Write down a capsulized program specification
   containing as a minimum the following:

   a) a statement of the problem
   b) a short description of the solution techniques involved
   c) summarized input requirements
   d) summarized output requirements
   *e) *testing/validation requirements*
   *f) performance requirements

2) Following the design and declaration of each variable, array, and structure include a comment defining its global asserted attributes.

   Example

```
DCL  I  FIXED  BIN (15,0);
/*ASSERT VALUE (I) (0:8) */
DCL A(1:100) FIXED BIN (15,0);
/* ASSERT VALUE (A(1:8)) (0,1)  */
/* ASSERT SUBSCRIPTRANGE (A(1:8))*/
```

3) It is suggested that assertions be used to test entry and exit conditions for all internal procedures.

4) It is further recommended that they be applied at entry/exit to internal algorithms within individual procedures (e.g., before and after important loop structures or procedure calls).

5) Operations or key algorithmic logic subject to possible singularities should be proceeded by appropriate assertions (e.g. A=B/Z where Z has been calculated previously should be proceeded by an assertion checking for Z not equal to 0).

6) Use as desired for debugging and other testing.

A Sample Program

Specification

Problem:

Create a program to solve the 8-Queens problem.

Description:

Given an 8x8 chessboard and 8 hostile queens. Find a position for each queen (or a total board configuration) where no queen may capture any other queen.

Inputs:

. None

Output:

. 8x8 chessboard configuration describing the placement of the 8 hostile queens

Test/Validation Criteria

. data integrity (ASSERT 8x8 board, ASSERT # QUEENS 8)

. every row, column contains 1 queen

. diagonals, contain at most one queen

Performance Criteria

. solution should be found in 20 cpu seconds on an IBM 360/91.

Program Development

Stepwise generation of partial solutions using a top-down development scheme might produce the following version of a program:

```
QUEENS: PROC;
   DECLARE
        board,pointer,safe;
   data_integrity_assertions;   /* LEVEL (2) */
   consider_first_column;
```

```
    LOOP: DO UNTIL (last_column_done          V    regress_out_of_first_col);
            try-column;
            IF safe
              THEN
                 DO;
                    setqueen;
                    consider_next_column;
                 END;
               ELSE
                 regress;
    END LOOP;
     algorithmic_validation_assertions;  /* LEVEL (4)      */
END QUEENS;
```

Although the final solution differs in many respects it basically follows N. Wirth's development of this program in his article "Program Development by Stepwise Refinement", CACM, Vol. 14, No. 4 (1971).

```
QUEENS: PROC;
      DECLARE          I          FIXED BIN (15,0);
        /* ASSERT                 VALUE (I) (0:8) LEVEL (2)  */
      DECLARE          J          FIXED BIN (15,0);
        /* ASSERT VALUE (J) (0:9) LEVEL (2)                  */
      DECLARE X(1:8) FIXED BIN (15,0);
        /* ASSERT VALUE (X(1:8) ) (0:8) L(2)          */
      DECLARE (A(1:8), B(2:16), C(-7:7) )   FIXED BIN (15,0)
          INIT( (8) 0,(15) 0,(15) 0);
        /* ASSERT VALUES (A(*), B(*), C(*) ) (0,1)L(2)  */
      DECLARE SAFE FIXED BIN(15,0)
        /* ASSERT VALUES(SAFE) (0,3)L(2)  */
      J=1;
      I=0;
      DO UNTIL (J > 8 | J< 1);
        DO UNTIL (SAFE =0 | I=8);
          I=I+1;
          SAFE=A(I)+B(I+J)+C(I-J);
        END;
          IF SAFE=0
            THEN
              BEGIN
                  /* ASSERT LOCAL VALUE (I,J)(1:8) LEVEL(5)  */
                  /* ASSERT LOCAL VALUES(A(I),B(K+J),C(I-J) ) (0) L(5)  */
                A(I) = A(I)+1;       /*                                  */
                B(I+J) = B(I+J)+1;  /* SET QUEEN                         */
                C(I-J) = C(I-J)+1; /*                                   */
                X(J)=I;
                J=J+1;
                I=0;
              END;
            ELSE
              CALL REGRESS (I,J,A,B,C,X);
      END;
        /* ASSERT LOCAL VALUES X(*) ) (1:8) LEVEL(4)          */
        /* ASSERT (X(II) ¬= X(JJ)) FOR ALL (II, JJ) (1:8) WHERE
            (II ¬= JJ) L(4)                                    */
        /* ASSERT LOCAL VALUES A(*) (1) L(4)                  */
        /* ASSERT LOCAL VALUES (B(*),C(*) ) (0,1) LEVEL (4)   */
        IF J>8
          THEN
            PUT DATA (X);
          ELSE
            PUT LIST ('FAILURE');
        STOP;
END QUEENS;
```

17950

```
REGRESS:  PROC(I, J, A, B, C, X);
    DECLARE(I, J) FIXED BIN (15, 0);
    /* ASSERT VALUE (I) (0:8) LEVEL(3)      */
    /* ASSERT VALUE (J) (0:9) L(3)          */
    DECLARE (A(1:8), B(2:16), C(-7:7) ) FIXED BIN (15,0);
    /* ASSERT VALUE (A(*), B(*), C(*) ) (0, 1) L(3)  */
    BEGIN;
     J=J-1;      /*  RECONSIDER_PRIOR_COL      */
     IF¬(J<1)  /* ¬REGRESS_OUT_OF_FIRST-COL  */
      THEN
       BEGIN;
       /* ASSERT LOCAL VALUE (I, J) (1:8) LEVEL(6) */
       /* ASSERT LOCAL VALUE (A(I), B(I+J), C(I-J) ) (1) L(6) */
       I=X(J);       /*
       A(I)=0;       /*                    */
       B(I+J)=0;     /*REMOVE QUEEN  */
       C(I-J)=0;     /*                    */
       IF       I = 8
          THEN
           BEGIN
            J=J-1;      /*  RECONSIDER_PRIOR_COL    */
            IF¬(J<1)  /*¬REGRESS_OUT_OF_FIRST_COL */
             THEN
              DO;
              /*  ASSERT LOCAL VALUE (I, J)(1:8) LEVEL (6) */
              /*  ASSERT LOCAL VALUE (A(I), B(I+J), C(I-J)(1)L(6)  */
              I=X(J);       /*                    */
              A(I)=0;       /*                     */
              B(I+J)=0;     /*  REMOVE QUEEN  */
              C(I-J)=0;     /*                    */
              END;
           END;
      END;
   END;
END REGRESS;
```

17949

REFERENCES

1. Dijkstra, E. W., "The Humble Programmer", 1972 Turing Award Lecture, Communications of the ACM, October 1972.

2. Dahl, O. J., Dijkstra, E. W., & Hoare, C.A.R., Structured Programming, Academic Press, London and New York, 1972.

3. Mills, H. D., "Mathematical Foundations for Structured Programming", Federal Systems Division, IBM, Gaithersburg, Md. FSC 72-6012, February 1972.

4. Dijkstra, E. W., Regional Conference on Programming Methodology, (private conversation), Albuquerque, New Mexico, January 7-11, 1974.

5. Wirth, N., Systematic Programming An Introduction, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1973.

6. Horowitz, E., et.al., "Practical Strategies for Developing Large Software Systems", Addison Wesley Publishing Co., Reading, Mass., May 1975.

7. Estrin, G., et al., "SNUPER COMPUTER - A Computer In Instrumentation Automation", AFIPS Spring Joint Computer Conference, 1967.

8. Russell, E. C., and Estrin, G., "Measurement Based Automatic Analysis of FORTRAN Programs", AFIPS Spring Joint Computer Conference, 1969.

9. Ingnalls, Daniel H. H., "FETE A FORTRAN Execution Time Estimator", Computer Science Department, Stanford University, (February 1971), STAN-CS-71-204.

10. Joseph, G. W., "The FORTRAN Frequency Analyzer as a Data Gathering Aid for Computer System Simulation", Directorate of Information Systems Technology, Deputy for Command and Management Systems, Electronic Systems Division, Air Force Systems Command, L. G. Hanscom Field, Bedford, Mass., March 1972.

11. Brown, J. R., et al., "Automated Software Quality Assurance: A Case Study of Three Systems", ACM SIGPLAN Symposium on Computer Program Test Methods, Chapel Hill, N. C., June 21-23, 1972.

12. Brown, J. R. and Hoffman, R. H., "Evaluating the Effectiveness of Software Verification - Practical Experience with an Automated Tool," AFIPS Fall Joint Joint Computer Conference, Anaheim, Calif., December 1972.

13. Stucki, L. G., "A Prototype Automatic Program Testing Tool", AFIPS Fall Joint Computer Conference, Anaheim, Calif., December 1972.

14. Stucki, L. G., "Automatic Generation of Self-Metric Software", Proceedings of the 1973 IEEE Symposium on Computer Software Reliability, New York City, April 30-May 2, 1973.

15. Stucki, L. G., and Svegel, N. P., "Software Automated Verification System Study," McDonnell Douglas Document No. MDC G5103, dated January 1974.

16. Stucki, L. G., "Tools Lessons Learned - New Strategies", Computer Science and Statistics: Eighth Annual Symposium on the Interface, Los Angeles, February 1975.

17. Stucki, L. G., and Foshee, G. L., "New Assertion Concepts for Self-Metric Software Validation", 1975 International Conference on Reliable Software, sponsored by ACM, IEEE, and NBS, Los Angeles, 21-23 April 1975.

18. Stucki, L. G., "New Research Areas in Automated Testing Tools", presented to Software Reliability Work Group JLC Electronic Systems Reliability Workshop, Arlie, Virginia, May 1975.

19. Stucki, L. G., "Testing Impact on the Future of Software Engineering", Fourth Texas Conference on Computing Systems, Austin, Texas, 17-18 November 1975.

20. Boettcher, C. B., "Program Evaluator and Tester", PET CDC Users Manual, McDonnell Douglas Automation Company, Document No. M2085074, 1974.

21. Stucki, L. G., et al., Workshop on Currently Available Program Testing Tools, Panel Discussion Los Angeles, 24-25 April 1975.

22. Howden, W. E. and Stucki, L. G., "Final Report Methodology for Effective Test Case Selection - Phase III, McDonnell Douglas Report to the National Bureau of Standards Contract #5-35933, January 1976.

APPENDIX A - SAMPLE PET OUTPUT

Overview

The Program Evaluator and Tester (PET) is a package of programs designed as an automated aid to assist in the debugging, testing and documenting of computer programs. The preprocessor module of PET operates on a FORTRAN source deck, gathering information on source statement characteristics and inserting FORTRAN code for automatically gathering run time statistics. It generates a modified FORTRAN program containing both the original source code and the inserted source code for later compilation and execution of an ordinary program case. Run time statistics are retrieved at program termination or under user control by a run time library. Reports are generated in a separate step by the PET postprocessor.

Run time statistics that may be gathered include statement execution and branch counts, min/max and first/last values of assignment statements and DO-loop parameters, detailed branching counts and relative subroutine execution timing. All but execution and branch counts are optional. Source statement characteristics gathered include percentages of executable and non-executable statements, number of comment statements and number of statements with ANSI standard FORTRAN violations which would affect portability of the program to other FORTRAN systems.

PET accepts as input any FORTRAN deck running on the host machine. No program modifications are necessary. PET is currently available on the CDC 6000 and 7000 series, the IBM 360 and 370 OS systems, and the UNIVAC 1100 series(ASCII FORTRAN compiler).

Sample PET Reports

The reports produced by the PET system present both detailed statement by statement statistics as well as program summary statistics.

a. Subroutine Listing and Execution Counts

This contains specific statement execution data as well as a profile of the subroutine. The execution and branch counts are matched back with the original source code and presented in the form of an annotated source listing.

First, the subroutine source is listed. An "N" is printed to the left of any statement that violates standard FORTRAN. To the right is the specific execution data: execution count for that statement, min/max and first/last values for assignment statements (if those options were indicated), and specific branching information for applicable statements. True/false counts are reported for logical IF statements and branch counts for arithmetic IF and assigned or computed GO TO statements.

Examples of the report for branching statements follow:

original statement	execution count	true count	false count
IF (A . EQ. B) GO TO 100	100	TRUE 70	FALSE 30

		# of transfers to 100	# of transfers to 200
GOTO (100, 200, 300), N	30	BRANCH1 5	BRANCH2 12
		BRANCH3 13	
		# of transfers to 300	

Figure A-1 illustrates a subroutine listing and execution count page.

```
                PROGRAM LISTING (LEADING N INDICATES CONVERSION WARNINGS)                COUNT        SPECIFIC EXECUTION DATA

      SUBROUTINE INPUT                                                                  02107
C                                                                                       02108
      REAL IFVD, MACHM, MACHT                                                           02109
C                                                                                       02110
      COMMON /GGG/ G(100)                                                               02111
C                                                                                       02112
      COMMON /ENT/ IENTRY                                                               02113
      COMMON /CONST/ DATA(100)                                                          02114
      EQUIVALENCE ( DATA(1) , DTP      )                                                02115
C                                                                                       02116
C                                                                                       02117
      NAMELIST /IN/ THETA,PSI,PHI,VM,GAMMA,SIGMA,XG,YG,ZG,ENDTME,                       02118
     *   ENDSTG,PI1,PT1,PI2,PT2,PI3,PT3,PI4,PT4,PI5,PT5,VT,XT,YT,ZT,                   02119
     *   TLON,YSTA,IFVD,PSIT,THETAT,AMODE,DT,AXT,AYT,AZT,                              02120
N    *   PP, QQ, RR, MACHM, MACHT, P, GMA                                              02121
C                                                                                       02122
      GO TO (100, 200, 300, 400, 500, 600, 700), IENTRY                                 02123        5  BRANCH 1      2    BRANCH 2      2
                                                                                                      BRANCH 3      0    BRANCH 4      0
                                                                                                      BRANCH 5      0    BRANCH 6      0
                                                                                                      BRANCH 7      1

  100 CONTINUE                                                                          02124        2
      RETURN                                                                            02125        2
  200 CONTINUE                                                                          02126        2
N     READ(5, IN, END=800)                                                            02127        2
N     PRINT 1                                                                         02128        2
    1 FORMAT(1H1)                                                                     02129
N     WRITE (6,IN)                                                                    02130        2
      G(1)=THETA                                                                      02131        2  MIN= 0.0              MAX= 0.0
                                                                                                      FIRST= 0.0            LAST= 0.0
      G(2)= PSI                                                                       02132        2  MIN= 0.0              MAX= 0.0
                                                                                                      FIRST= 0.0            LAST= 0.0
      G(3)= PHI                                                                       02133        2  MIN=-0.45000000E 02   MAX=-0.45000000E 02
                                                                                                      FIRST=-0.45000E 02    LAST= -0.45000E 02
      IF ( VM .GT. 0.)  GO TO 210                                                     02134        2  TRUE          1       FALSE         1
      CALL ATMOS ( -ZG, PA, VS )                                                      02135        0
      G(4) = VS * MACHM                                                               02136        0
      GO TO 211                                                                       02137        0
  210 CONTINUE                                                                        02138        1
      G(4) = VM                                                                       02139        1  MIN= 0.75420000E 03   MAX= 0.75420000E 03
                                                                                                      FIRST= 0.75420E 03    LAST= 0.75420E 03
  211 CONTINUE                                                                        02140        1
      G(5)=GAMMA                                                                      02141        1  MIN= 0.0              MAX= 0.0
                                                                                                      FIRST= 0.0            LAST= 0.0
      G(6)=SIGMA                                                                      02142        1  MIN= 0.0              MAX= 0.0
                                                                                                      FIRST= 0.0            LAST= 0.0
      G(7)=XG                                                                         02143        1  MIN= 0.0              MAX= 0.0
                                                                                                      FIRST= 0.0            LAST= 0.0
      G(8)=YG                                                                         02144        1  MIN= 0.0              MAX= 0.0
                                                                                                      FIRST= 0.0            LAST= 0.0
      G(9)=ZG                                                                         02145        1  MIN=-0.10000000E 05   MAX=-0.10000000E 05
                                                                                                      FIRST=-0.10000E 05    LAST= -0.10000E 05
      G(39)=ENDTME                                                                    02146        1  MIN= 0.30000000E 02   MAX= 0.30000000E 02
                                                                                                      FIRST= 0.30000E 02    LAST= 0.30000E 02
```

SPECIFIC BRANCHING INFORMATION

LOGICAL IF BRANCHES

NONSTANDARD FORTRAN

STATEMENTS NOT EXECUTED

Figure A-1

Copy available to DDC does not permit fully legible reproduction

b. Subroutine Summaries

Following the listing is the subroutine profile. The syntactic profile gives the total number of statements of all kinds in the subroutine and then lists statements of various types,giving the number and percent of each in the subroutine.

The statement types and their PET definition are:

Executable source	All FORTRAN statements except comment and nonexecutable
Nonexecutable source	COMMON,DIMENSION, Type statements, DATA,EQUIVALENCE, OVERLAY Control Cards, STATEMENT FUNCTIONS, FORMAT
Comment	Statements with 'C' in Column 1
Nonstandard	Statements that violate standard FORTRAN (Figure A-1)
BRANCH	Total of all GOTO and IF branches
CALL	All subroutine CALL statements. Function subroutine references are not delineated.
UNFORMATTED I/O	All binary read and write statements
FORMATTED I/O	All read and write statements with a format

The operational profile gives the total execution count for the subroutine and then, by statement type, lists the number of statements that were executed and the percent that were executed. For branch(i.e. GOTO and IF statements), it gives the number of branches that were taken and the percentage of the possible branches that were taken.

Figure A-2 shows a typical summary report.

PROGRAM EVALUATOR AND TESTER REPORT
FOR SUBROUTINE INPUT

SYNTACTIC PROFILE

TOTAL NUMBER OF SOURCE STATEMENTS 103

TYPE OF STATEMENT	NUMBER	PERCENT OF TOTAL
EXECUTABLE SOURCE	63	60.6
NONEXECUTABLE SOURCE	9	8.7
COMMENT	11	10.7
NONSTANDARD	4	3.9
BRANCH	20	N/A
CALL	3	N/A
UNFORMATTED I/O	0	N/A
FORMATTED I/O	3	N/A

PERCENT OF STATEMENTS THAT ARE EXECUTABLE

STATIC PROFILE OF SUBROUTINE BY STATEMENT TYPE

PERCENT OF POSSIBLE SOURCE STATEMENTS EXECUTED

PERCENT OF BRANCHES EXECUTED

OPERATIONAL PROFILE

TOTAL EXECUTION COUNT 77

TYPE OF STATEMENT	NO EXECUTED	PERCENT EXECUTED
EXECUTABLE SOURCE	63	75.9
BRANCH	10	50.0
CALL	1	33.3
UNFORMATTED I/O	0	100.0
FORMATTED I/O	3	100.0

DYNAMIC PROFILE OF SUBROUTINE EXECUTION BY STATEMENT TYPE

Figure A- 2

c. Program Summary

The information on this page has been compiled from the subroutine profile into a composite profile of the entire program. It includes a program syntactic profile and an operational profile in the format described in the previous section.

d. Subroutine Operational Summary

This page summarizes selected subroutine operational statistics. The information gives a picture of how well the case(s) that were executed actually exercised the program branches. It can be used by the analyst to design additional test cases to exercise other areas of code or to determine that some areas of the code can never be executed.

Figure A-3 illustrates a subroutine operational summary. The following explains the information on the report by column.

For each subroutine that was executed the following information is given.

Column	Information
A	Subroutine name
B	Number of executable statements that were actually executed
C	Percent of the total executable statements that were actually executed. Refer to the subroutine summary page for the total number of executable statements.
D	Number of subroutine calls that were actually executed.
E	Percent of the subroutine calls that were executed. Refer to the subroutine summary page for the total number of call statements.
F	Number of branches that were actually taken.
G	Percent of the possible branches that were actually taken.

The above information is then totaled and combined with the syntactic statistics for the subroutines that were not executed to give the program operational summary (H).

Following the summary is a listing of the names of the subroutines that were instrumented but not executed (I).

PROGRAM EVALUATION AND TESTER REPORT

SUBROUTINE OPERATIONAL SUMMARY

(A) SUBROUTINE NAME	(B) NO OF STATEMENTS EXECUTED	(C) PERCENT EXECUTED	(D) NO OF CALLS EXECUTED	(E) PERCENT EXECUTED	(F) NO OF BRANCHES EXECUTED	(G) PERCENT EXECUTED
(MAIN)	39	100.0	13	100.0	5	55.6
AUTS	69	95.8	0	100.0	9	81.8
AWCRFT	21	84.0	0	100.0	5	71.4
ATMOS	21	70.0	0	100.0	8	42.1
CHECK	15	57.7	0	100.0	12	50.0
CLIP	2	100.0	0	100.0	0	100.0
CNTRL	115	60.8	7	35.0	39	50.6
DERIV	77	78.6	42	71.2	8	66.7
EOOM	35	89.7	0	100.0	5	71.4
EXPLT	1	100.0	0	100.0	0	100.0
PLOT1	1	100.0	0	100.0	0	100.0
FCC	70	83.3	1	100.0	6	30.0
GLTOF	119	96.7	4	100.0	31	93.9
INIAL	46	85.2	1	100.0	3	42.9
INPUT	63	75.9	1	33.3	10	50.0
INVRS	16	100.0	0	100.0	0	100.0
LOOK1	5	100.0	1	100.0	0	100.0
LOOK2	9	100.0	2	100.0	0	100.0
LOOK3	15	100.0	3	100.0	0	100.0
M3X1	6	100.0	0	100.0	0	100.0
PRINT	31	72.1	0	100.0	1	14.3
RELT	83	91.2	0	100.0	3	42.9
RKS4	125	85.0	8	80.0	32	45.7
SACS	112	96.6	10	100.0	40	90.9
TARGET	31	88.6	0	100.0	6	66.7
TMULT1	6	100.0	0	100.0	0	100.0
TFFS	20	83.3	1	100.0	8	72.7
VPCS	19	82.6	1	100.0	6	66.7
(H) PROGRAM	1172	82.5	95	73.6	237	57.9

CUMULATIVE ~~STATISTICS~~ FOR PROGRAM

(I) SUBROUTINES NOT EXECUTED

BLKDAT
BLKDAT.
BLKDAT.
GRAFIX
PLOT1
STAGE

THESE SUBROUTINES WERE INSTRUMENTED BUT NOT EXECUTED. THE THREE NAMED BLKDAT. WERE BLOCK DATA ROUTINES.

Figure A-3

e. Subroutine Execution Summary

This page summarizes the execution data for all the monitored subroutines. It gives a profile of the types of operations actually performed by the subroutine (i.e., is it mainly performing I/O or logic or calling other subroutines). The information on this page combined with the timing information described in the next section can help the analyst determine where a program is spending its time and how it could be improved. Figure A-4 shows a subroutine execution summary. The information on the report is explained by column below.

Column	Information
A	Subroutine name
B	Number of times the subroutine was called
C	Total execution count = sum of the execution counts for each statement
D	Total call count = sum of the execution counts for each call statement
E	The percent of the executions that were subroutine calls (i.e., total call count/total execution count)
F	Total branch count = sum of the execution counts for each branch statement
G	The percent of the executions that were branches (i.e., total branch count/total execution count)
H	Total input/output count = total execution count for formatted and unformatted read and write statements
I	The percent of the executions that were I/O (i.e., total input-output count/total execution count).

PROGRAM EVALUATOR AND TESTER REPORT

SUBROUTINE EXECUTION SUMMARY

(A) SUBROUTINE NAME	(B) NUMBER TIMES ENTERED	(C) TOTAL EXECUTION COUNT	(D) TOTAL CALL COUNT	(E) PERCENT OF TOTAL EXECUTIONS	(F) TOTAL BRANCH COUNT	(G) PERCENT OF TOTAL EXECUTIONS	(H) TOTAL I/O COUNT	(I) PERCENT OF TOTAL EXECUTIONS
(MAIN)	1	679	18	2.65	5	0.74	0	0.0
AUTS	970	50198	0	0.0	2880	5.70	21	0.04
ARCHFT	970	2940	0	0.0	970	32.99	21	0.71
ATMOS	946	17979	0	0.0	1893	10.53	0	0.0
CHECK	17955	90111	0	0.0	88	0.10	0	0.0
CLIP	15146	30292	0	0.0	0	0.0	0	0.0
CNTRL	106	9151	246	2.69	2949	32.23	102	1.11
DERIV	970	34293	9650	28.14	1915	5.58	0	0.0
ECOM	970	25590	0	0.0	970	3.79	0	0.0
EXPLT	1	1	0	0.0	0	0.0	0	0.0
PLOT1	102	102	0	0.0	0	0.0	0	0.0
RCC	1	70	1	1.43	6	8.57	11	15.71
SUIDE	970	47258	968	2.05	11580	24.50	63	0.13
INTAL	4	51	1	1.96	4	7.84	0	0.0
INPUT	5	77	1	1.30	12	15.58	6	7.79
INVR3	102	2652	0	0.0	0	0.0	0	0.0
LOOK1	2835	17010	2835	16.67	0	0.0	0	0.0
LOOK2	9725	58700	9450	16.67	0	0.0	0	0.0
LOOK3	1890	83160	5670	6.82	0	0.0	0	0.0
M3X1	1	18	0	0.0	0	0.0	0	0.0
PRINT	21	651	0	0.0	21	3.23	315	48.39
RELT	967	75758	0	0.0	967	1.28	21	0.03
RKS4	1	347186	1047	0.30	1943	0.56	0	0.0
SACS	970	80067	7560	9.40	15793	19.63	21	0.03
TARGET	970	9558	0	0.0	1915	20.04	21	0.22
TMULT1	945	17010	0	0.0	0	0.0	0	0.0
TFFS	970	10472	945	9.02	2960	27.31	0	0.0
VPCS	970	9527	945	9.92	1915	20.10	0	0.0

Figure A-4

f. Subroutine Timing Summary

The statistics for subroutine timing are gathered by accessing the system clock at all entries into and exits from a subroutine. The accumulation of timing statistics is turned off when a subroutine call is made and when a user-supplied function subroutine is referenced in the first term of an assignment statement. Note that timing is not turned off when a function subroutine is referenced in a IF statement, thus the time spent in the function will be included in the total time of the calling subroutine. In order to obtain the most accurate timing history, it is recommended that function references be made in separate assignment statements. Note also that time spent in FORTRAN-supplied functions and in system routines will be accumulated in the calling routine.

The timing history obtained is distorted by the PET instrumentation. For this reason, it is recommended that when an accurate timing history is desired that the min/max and first/last option not be used. However, experience has shown that the relative time spent in subroutines is not significantly distorted by the PET instrumentation.

A function subroutine reference will be considered to be user-supplied unless:

1) it is in a table of FORTRAN-supplied functions
2) it is a FORTRAN intrinsic function, but was referenced in an external or type statement.

The various types of timing information output on the subroutine timing summary are enumerated below.

- The total execution time for the case is the elapsed time from entry into main routine to exit from the instrumented programs.
- The total time in monitored routines is the total time spent in routines for which option 32 was turned on.
- The total time in other routines is the difference between the first two numbers.
- The histogram of subroutine timing is output for those subroutines for which the timing option is turned on and which accumulated sometime on the clock.
- The actual time and the percent of monitored time spent in each of the above subroutines is output to the right of the histogram.
- The elapsed time spent in the postprocessor is output.

If the timing option was not used, only the total execution time for the case and the total time in the postprocessor will be given.

Figure A-5 shows a typical timing summary. Subroutine XUB accumulated the largest amount of time, so it was printed with 50 asterisks. The actual time spent in XUB was 31.926 seconds and the percent of monitored time was 31/71=35%. Subroutine PCUM accumulated 25.709 seconds so it has 40 asterisks = 25.705/31.926x50.

PROGRAM EVALUATOR AND TESTEM MEMORY
SUBROUTINE TIMING SUMMARY

TOTAL EXECUTION TIME FOR THIS CASE 161,271 SEC

TOTAL TIME IN MONITORED ROUTINES 89,644 SEC
TOTAL TIME IN OTHER ROUTINES 71,627 SEC

} TOTAL TIME SPENT IN ROUTINES WHERE TIMING WAS TURNED ON VERSUS TIME SPENT IN ALL OTHER SUBROUTINES.

DETAILED SUBROUTINE TIME MONITORING

SUBROUTINE NAME	TIME RELATIVE TO MAXIMUM SUBROUTINE EXECUTION TIME (• = TWO PERCENT)	ACTUAL TIME IN SECONDS	PERCENT OF TOTAL MONITORED TIME	
LINKOD		,002	,00	
DETCTR		,008	,01	
PLOM	••••••••••••••••••••••••••••••••••••••	25,709	28,67	
PISCTR	••••••••••••	7,706	8,60	
RLP73	•••••••••••••••••••••••••••••••	23,088	25,75	
XOR	•••	31,929	35,61	← THIS SUBROUTINE ACCUMULATED THE MOST CLOCK TIME.
SD52		,009	,01	
SUMN		,313	,35	
RADAR	•	,829	,92	
TMS674		,002	,00	

TOTAL TIME IN POSTPROCESSOR 1,203 SEC

Figure A-5

APPENDIX B

SAMPLES FROM A PROPOSED STRUCTURED FORTRAN ASSERTION SYSTEM

SAMPLE REPORTS

ORIGINAL SOURCE CODE LISTING

EXECUTION COUNT

ASSERTION INFORMATION
BRANCH COUNTS, TRUTH VALUES,
DATA RANGES

		ANNOTATED PROGRAM LISTING	COUNT	SPECIFIC EXECUTION DATA
00001		PROGRAM TICTAC (INPUT,OUTPUT)		
00002	C			
00003	C	INTERACTIVE TIC TAC TOE PLAYER		
00004	C			
00005		INTEGER FIRST, GAMES, MOVE, PLAYER, TURNS, WINNER		
00006		INTEGER HUMAN, MACHINE, NOBODY, QUIT, BOARD(9), NAME(10		
00007		COMMON /CONST/ HUMAN, MACHINE, NOBODY, QUIT, NAME		
00008	C			
00009	C	GLOBAL TRACE 10 VIOLATIONS		
00010	C	ASSERT VALUES(TURNS)(0,5)		
00011	C	MONITOR RANGE FIRST,LAST(PLAYER,MACHINE,TURNS)		
00012	C	MONITOR CHARACTER RANGE FIRST,LAST(NAME(*))		
00013	C			
00014		DATA NOBODY/0/, MACHINE/2/, HUMAN/1/, QUIT/6/		
00015		DATA NAME/' ','HTL','HTC','HTR','HML','HMC','HMR','HBL','HBC','HBR		
00016		X 'HRO'/		
00017		CALL PRINTRO	1	
00018	C	PLAY SOME GAMES		
00019		FIRST = HUMAN	1	
00020		GAMES = 0	1	
00021		LOST = 0	1	
00022		MYSCORE = 0	1	
00023		WHILE WINNER .NE. QUIT DO	1	
00024	C	PLAY 1 GAME		
00025		GAMES = GAMES+1	3	
00026		WINNER = NOBODY	3	
00027		TURNS = 0	3	
00028		FOR I=1 STEP 1 UNTIL 9 DO	3	
00029		BOARD(I) = NOBODY	27	
00030		ENDDO	27	
00031		IF FIRST.EQ.MACHINE THEN	3	TRUE 1 FALSE 2
00032		BOARD(5) = MACHINE	1	
00033		TURNS = 1	1	
00034		PRINT 00005, NAME(5)	1	
00035	00005	FORMAT(' HAVE SELECTED SQUARE ',A2)		
00036		ENDIF	3	
00037		WHILE WINNER.EQ.NOBODY .AND. TURNS.LT.9 DO	3	
00038		PLAYER = HUMAN	10	MIN= 3 MAX= 3 FIRST= 3 LAST= 3
00039		CALL HUMNMOV(BOARD,MOVE)	10	
00040		IF MOVE.NE.10 THEN	10	TRUE 9 FALSE 1
00041		BOARD(MOVE) = HUMAN	9	
00042		TURNS = TURNS+1	9	MIN= 1 MAX= 9 FIRST= 1 LAST= 3

ASSERTION VIOLATIONS 4
ASSERT VALUES TURNS:(0,5)

EXEC NUMBER	VALUE
3	6
4	7
5	8
6	9

SAMPLE REPORTS

ORIGINAL SOURCE CODE LISTING

EXECUTION COUNT

ASSERTION INFORMATION BRANCH COUNTS, TRUTH VALUES, DATA RANGES

```
                ANNOTATED PROGRAM LISTING                    C OUNT   SPECIFIC EXECUTION DATA

00043             CALL CHEKWIN (BOARD, MOVE, TURNS, PLAYER, WINNER)    9
00044 C             TRACE 10 VIOLATIONS
00045 C               ASSERT (MOVE .LT. 9) HALT ON 10                  9   ASSERTION VIOLATIONS
                                                                             EXEC NUMBER  VARIABLE  VALUE
                                                                                    6     MOVE        9
                                                                                    7     MOVE        9
00046 C             TRACE OFF
0004/               IF WINNER . EQ . NOBODY . AND. TURNS . LT . 9 THEN   9   TRUE          8 FALSE     1
00048                 PLAYER = MACHINE                                   9   MIN=          2 MAX=      2
                                                                             FIRST=        2 LAST=     2
00049               CALL MACHMOV (BOARD, TURNS, MOVE)                    8
00050               BOARD(MOVE) = MACHINE                                8
00051               TURNS = TURNS + 1                                    8   MIN=          2 MAX=      8
                                                                             FIRST=        2 LAST=     4
                                                                             ASSERTION VIOLATIONS      3
                                                                             ASSERT VALUES ( TURNS) (0:5)
                                                                               EXEC NUMBER          VALUE
                                                                                    3                 6
                                                                                    4                 7
                                                                                    5                 8
00052               CALL CHEKWIN (BOARD, MOVE, TURNS, PLAYER, WINNNER)   8
00053             ENDIF                                                  9
00054           ELSE                                                     1
00055             WINNER = QUIT                                          1
00056           ENDIF                                                   10
00057         ENDDO                                                     10
00058         CALL PRESULT (BOARD, WINNER, GAMES, MYSCORE, LOST)         3
00059         IF FIRST . EQ. MACHINE THEN                                3   TRUE          1 FALSE     2
00060           FIRST = HUMAN                                            1
00061         ELSE                                                       2
00062           FIRST = MACHINE                                          2
00063         ENDIF                                                      3
00064       ENDDO                                                        3
00065       END                                                          1
```

140

SAMPLE REPORTS

AUTOMATED VERIFICATION SYSTEM
OPERATIONAL PROFILE
SUBROUTINE TICTAC

DATE 01/31/75

TYPE OF STATEMENT	NUMBER	NUMBER EXECUTED	PERCENT EXECUTED
TRANSFER			
IF	4	4	100.0
TRUE CONDITIONS		4	100.0
FALSE CONDITIONS		4	100.0
CALL	6	6	100.0
GO TO	0	0	N/A
CASE	0	0	N/A
ALSO CONDITIONS	0	0	N/A
OTHERWISE CONDITIONS	0	0	N/A
DO	3	3	100.00
WHILE CONDITIONS	2	2	100.00
ASSIGNMENT	19	19	100.00
I/O			
READ	0	0	N/A
WRITE	1	1	100.00
OTHER EXECUTABLE SOURCE	9	9	100.00

SAMPLE REPORTS

INSTRUMENTATION SUMMARY

LOCAL ASSERTIONS

STMT NO.	ASSERTION	EXEC COUNT	VIOLATIONS
00045	ASSERT (MOVE .LT. 9) HALT ON 10	9	2

GLOBAL ASSERTIONS

ASSERT VALUES (TURNS) (0:5)

VARIABLE	STATEMENTS THAT CHANGE THE VALUE OF VARIABLE	STATEMENTS THAT CAUSED AN ASSERTION VIOLATION
TURNS	00027 00033 00042 00051	00042 00051

MONITOR RANGE FIRST LAST (PLAYER, MACHINE, TURNS)

VARIABLE	STATEMENTS THAT CHANGE THE VALUE OF VARIABLE	VALUES ATTAINED MIN	MAX
PLAYER	00038 00048	2	3
MACHINE	00014	2	2
TURNS	00027 00033 00042 00051	0	9

MONITOR CHARACTER RANGE FIRST LAST (NAME(*))

VARIABLE	STATEMENTS THAT CHANGE THE VALUE OF VARIABLE	VALUES ATTAINED MIN	MAX
NAME(*)	00015	'BC '	'TR '

SOFTWARE TESTING TOOLS

(JAVS AND RXVP)

W. R. Wisehart

August 1976

GENERAL RESEARCH CORPORATION
5383 HOLLISTER AVENUE • PHONE (805) 964-7724
P.O. BOX 3587, SANTA BARBARA, CALIFORNIA 93105

Presented at the Department of Defense/Industry Conference on Software Verification and Validation (V&V), August 3-5, 1976, Syracuse, New York.

Copy available to DDC does not
permit fully legible reproduction

INTRODUCTION

JAVS, for JOVIAL Automated Verification System, is an integrated set of software testing tools created to assist in testing computer programs written in the J3 dialect of JOVIAL. RXVP is a similar, but more extensive, set of tools for FORTRAN and IFTRAN programs. Both systems are derived from RSVP which was originally created by Dr. Edward F. Miller, Jr. and Dr. Michael Paige. JAVS design and RXVP development were major efforts of the GRC Program Validation Project while Dr. Miller was the leader of that group. The emphasis of this description will be JAVS, which is currently being applied to testing a large software package at RADC. Unfortunately, it is too early in that program to permit me to comment on the contribution of JAVS to the testing effort. The results of these tests will be published as an RADC report early next year.

In Fig. 1, I have partitioned the universe of software behavior in two ways: the specified and unspecified, and the acceptable and unacceptable. The collective experience with software development says that all four of these forms of behavior will still exist at the time software is declared ready for acceptance, and will continue to exist after acceptance testing is over, ~~primarily~~ partly because the testing process is confined to examining points in the vector space of the input, and partly because there are no [illegible] that distribute [illegible]

In a typical software test activity the testing group is attempting to map these regions by probing with single-point test cases. Their success is dependent on the total resources devoted to exploring the

Copy available to DDC does not
permit fully legible reproduction

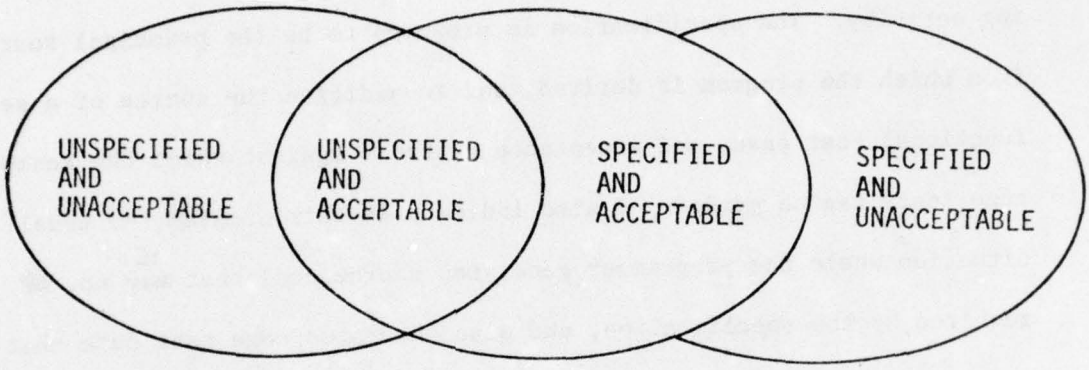


Figure 1. Universe of Software Behavior

universe of behavior and the effectiveness of those resources in terms of the location of the tested points in that universe. Test tools such as JAVS and RXVP are designed to increase the effectiveness of the testing process and to replace human resources with computer resources.

Figure 2 shows how an AVS such as JAVS or RXVP fits into the testing activity. The specification is presumed to be the principal source from which the program is derived, and in addition the source of a set of functional test cases and acceptance criteria against which the acceptance tests can be graded. I also indicate in this diagram the usual situation where the programmer generates source code that ~~may~~ is not required by the specification, and also generates some test data that exercises the capability so provided. All test cases that exist when the software is released for acceptance testing will undoubtedly lie in the acceptable region. The acceptance tester will be charged with generating new test cases (the supplied cases have already been learned by the software) that demonstrate acceptable behavior while spanning the universe of total behavior to identify the regions of unacceptable behavior.

There are five functions performed by the AVS in this role:

- Analysis of source code and creation of a library of module data
- Generation of reports based on static analysis of the source code that reveal existing or potential problems in the code, and identify the software structure

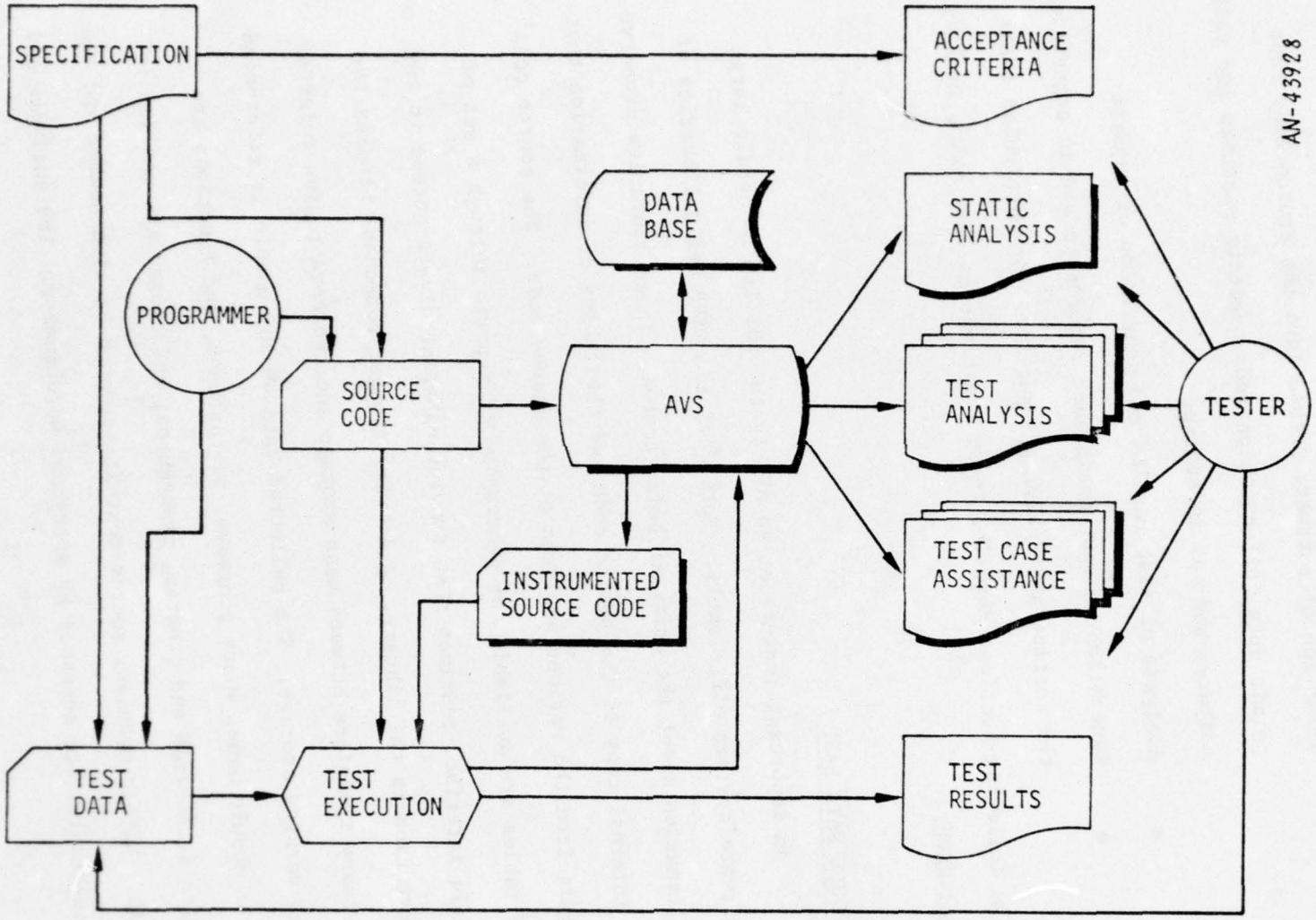


Figure 2. Software Analysis and Testing Augmented by AVS

- Insertion of software probes into the source code that permit data collection on code segments executed and values computed and set in storage
- Analysis of test results and generation of reports
- Generation of test assistance reports to aid in organizing the testing and deriving input sets for particular tests

The following sections describe these functions as they exist in JAVS and RXVP.

MODULE DATA BASE

An important feature of an AVS is the ability to handle large software systems efficiently. Both JAVS and RXVP build libraries of information about the software being tested. These libraries preserve an internal copy of the source code and the tables of information that result from the various analyses of the source code. The source code and tables are available for generation of reports through a set of simple interface routines that provide apparent direct access to any information in the library. A library manager component handles the necessary transfers between mass storage and the core region referred to as working storage. The collected data on any module is referenced by the module name, where programs, subroutines, and functions are modules in FORTRAN and program, procedures, and closes are modules in JOVIAL. Within modules, source text is accessed by line number and the various tables are accessed by entry and word number. The analysis and

reporting routines have a simple interface to all the source code of a large system, and this simple interface permits easy addition of new analyses and reports.

STATIC ANALYSIS

One approach to increasing software quality (i.e., reducing the region of unacceptable behavior) is to prohibit the use of certain coding practices that are prone to mistakes. Where the compilers do not enforce such standards, the AVS is a convenient technique for identifying violations of standards or clear mistakes in the programming. Static analysis also produces reports on module and system structure that would normally be produced one time as a form of documentation. A partial description of the available reports from both JAVS and RXVP is as follows:

- Multiple Module Cross Reference
  This report lists all symbols that appear in all modules of a library and for each module where the symbol appears gives the statement number and flags declarations, and the statement numbers of statements where the symbol is set. This report is shown in Fig. 3.

- Module Invocation Report
  This report lists all of the references to other modules for a single module in addition to all invocations of that module from other modules in the library giving the statement number and the text of the invocation. In RXVP this report also includes a comparison of the actual parameters of an invocation with the formal parameters of the module description (see Fig. 4).

- Module Dependencies
  There are four static reports that display module dependencies within a library or selected groups of modules on a library. The invocation tree is complete to the extent it can be derived from the library and would normally be produced with respect to the program that sits at the top of the hierarchy.

```
GENERAL CROSS REFERENCE LISTING

    FOR WHOLE LIBRARY

SYMBOL     MODULE     USED/SET/DEFINITION ( * INDICATES SET, D INDICATES DEFINITION )

BG         EXPROGM      18D   28
CARD       EXPROGM      12D   18*   21    22    23
EXMPL1     EXMPL1        1
           EXPROGM      24
EXMPL2     EXMPL2        1
           EXPROGM      27
EXMPL3     EXMPL1       29
           EXMPL3        1
FILL       EXMPL1        5D   15*
ID         EXPROGM       5D   15    21
INDXS      EXMPL1        7D   22*   24*   28
ITER1A     EXPROGM       8D   10    15    22
ITER1      EXMPL2        4*
           EXPROGM       6D   10D   24    26
ITER2A     EXPROGM       9D   11    15    23
ITER2      EXMPL2        5*
           EXPROGM       7D   11D   24
LAREL1     EXMPL1       27    27    27    30D
LABEL2     EXMPL1       27    29D
LIMIT1     EXMPL1        1     3D    9    10*   12
LIMIT2     EXMPL1        1     4D   18
MESAGE     EXCOMPL       5D
MESSAG     EXCOMPL      11D
           EXMPL1       31*   33*   35*   38
           EXPROGM      16*   17
MSG1       EXCOMPL       7D
           EXPROGM      16
MSG2       EXCOMPL       8D
           EXMPL1       31
MSG3       EXCOMPL       9D
           EXMPL1       33
MSG4       EXCOMPL      10D
           EXMPL1       35
PICK       EXMPL1       27D   28
PRINTR     EXMPL1       38
           EXPROGM      14D   17
READER     EXPROGM      13D   18    19
RESULT     EXMPL1        6D   11*   16*   16    30    32
           EXMPL3        4*
TAPE5      EXPROGM      13
TAPE6      EXPROGM      14
```

Figure 3. Multi Module Cross Reference Report

```
MODULE INVOCATION SPACE...
MODULE <NPUT    >, JAVSTEXT <NPUT    >, PARENT MODULE <NPUT    >
-------------------------------------------------------------------------------------------------------------
  3              PROC NPUT $                                                                    (   1)
-------------------------------------------------------------------------------------------------------------
INVOCATIONS FROM WITHIN THIS MODULE
-----------------------------------

MODULE BTOH
                    STMT =   68         BTOH ( LYNCNT )
                    STMT =   69         BTOH ( STCTR )
MODULE ERRXR
                    STMT =   29         ERRXR ( 4 )
MODULE GTCARD
                    STMT =   41         GTCARD ( = STATUS , CRD1 )
MODULE HOLS
                    STMT =   18         HOLS ( HDEF ($ A $) )
MODULE INMDB
                    STMT =   46         INMDB ( 0 )
MODULE PRXX
                    STMT =   79         PRXX ( HEL , 120 , EJCT )
MODULE TERMIN
                    STMT =   49         TERMIN ( 1 )
-------------------------------------------------------------------------------------------------------------
INVOCATIONS TO THIS MODULE FROM WITHIN LIBRARY
----------------------------------------------

MODULE GTCR
                    STMT =   31         NPUT
MODULE OLOOK
                    STMT =  119         NPUT
                    STMT =  306         NPUT
                    STMT =  363         NPUT
-------------------------------------------------------------------------------------------------------------
```

Figure 4. Dependence Invokes Report

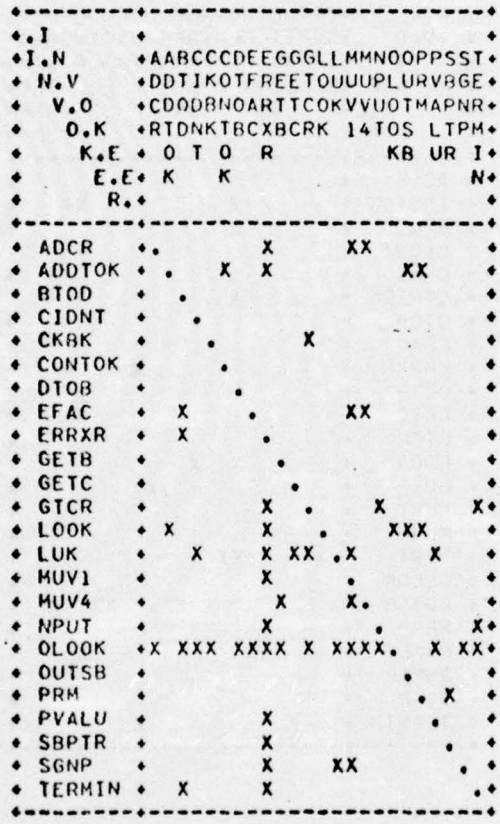
The other three reports are produced in matrix form and are shown in Figs. 5, 6, and 7. Figure 5 is an invocation matrix for all modules on a library. Figure 6 shows a similar matrix for all invocations from a library that aren't on the library data base. The report shown in Fig. 7 displays the internal interconnections of a selected group of modules as well as invocations to the group and invocations from the group.

The remaining functions of JAVS and RXVP are those that support testing of the execution. These components support automatic instrumentation, data collection, data analysis, and assistance in generating new test cases.

AUTOMATIC INSTRUMENTATION

Both JAVS and RXVP automatically insert software probes such that a record can be made of the statements and decision outcomes that have occurred during execution of the program. One form of instrumentation is associated with the smallest logical unit of code. This logical unit is identified in the source text as the sequence of statements lying between two decisions and is called a decision-to-decision path (DD-path). The instrumentation component of the AVS performs the required analysis, generates the text of the probe, adds it to the source code and, if desired, stores the instrumented module back on the library. An instrumented JOVIAL module is normally kept in source form on the library since it must be compiled with proper variable scope. Instrumented FORTRAN modules are typically saved as compiled object code.

LIBRARY DEPENDENCE TABLE



	I I.N N.V V.O O.K K.E E.E R.	AABCCCDEEGGGLLMMNOOPPSST DDTIKOTFREETOUUUPLURVBGE CDODBNOARTTCOKVVUOTMAPNR RTDNKTBCXBCRK 14TOS LTPM O T O R KB UR I K K N
ADCR	.	X XX
ADDTOK	. X X XX	
BTOD	.	
CIDNT	.	
CKBK	. X	
CONTOK	.	
DTOB	.	
EFAC	X . XX	
ERRXR	X .	
GETB	.	
GETC	.	
GTCR	X . X X	
LOOK	X X . XXX	
LUK	X X XX .X X	
MUV1	X .	
MUV4	X X.	
NPUT	X . X	
OLOOK	X XXX XXXX X XXXX. X XX	
OUTSB	.	
PRM	. X	
PVALU	X .	
SBPTR	X .	
SGNP	X XX .	
TERMIN	X X .	

Figure 5. Dependence Library Report

AUXILIARY LIBRARY DEPENDENCE TABLE

.I	ABEFGGGHIIMMMMOPPPRRSSW
I.N	CTRAETVONNDDDUPPUUETDYR
N.V	SURTTCELMSBBBVUXITMSBSA
V.O	THOABAUSDBNNS2TXBLQ STP
O.K	B RLLRP B AET LSU TEU
K.E	KD MX KTO RMP
E.E	P
R.	
ADCR	
ADDTOK	X XX
BTOD	
CIDNT	X
CKRK	
CONTOK	X X
DTOB	
EFAC	
ERRXR	X X X
GETB	X
GETC	X
GTCR	
LOOK	X X
LUK	X
MUV1	
MUV4	
NPUT	X X XX X
OLOOK	
OUTSB	X X X XX XX X
PRM	
PVALU	X X
SBPTR	
SGNP	
TERMIN	X X X

Figure 6. Dependence Auxlib Report

GROUP DEPENDENCY TABLE

```
+--------+--------+-----------------------------+
+.I       +        +                             +
+I.N      +ACGGOOPS+AABCCDEEGGHILMMMMNPPPRRSST+
+ N.V     +DOEELURB+CDTIKTFRETONUDDUUPUUVETDGE+
+  V.O    +DNTTOTMP+SCODBOARTCLSKBBVVUTTAMSBNR+
+   O.K   +TTBCOS T+TRDNKBCXBRSB NS14TBLLQ SPM+
+    K.E  +OO  KB R+B  T   RL    ET   LSUU T I+
+     E.E +KK      +        K    X    KT O R N+
+      R. +        +                             +
+--------+--------+-----------------------------+
+ ADDTOK  +.X   XX +       X  X           XX    +
+ CONTOK  + .      +          X           X     +
+ GETB    +  .     +                      X     +
+ GETC    +   .    +                      X     +
+ OLOOK   +  X .   + XXXXXXX X  X  XXX  X    XX+
+ OUTSB   +     .  +X        X  X XX   XX   X   +
+ PRM     +      .X+                            +
+ SBPTR   +       .+       X                    +
+--------+--------+-----------------------------+
+ LOOK    +X   XXX +       X  X                 +
+ LUK     +  X     +   X   X X     X     X      +
+ MUV4    + X      +                X           +
+--------+--------+-----------------------------+
```

Figure 7. DEPENDENCE,GROUP,MODULES=...

In addition to the decision-to-decision path instrumentation, RXVP provides full parameter instrumentation in which case data is collected on each variable as it is set, on DO loop parameters as used, and on logical values at decisions. JAVS allows the user to insert instrumentation through which data is collected on all values set into selected variables or on values which do not fall within a range definition.

TEST EXECUTION AND DATA ANALYSIS

In test execution the instrumented source code generated by the AVS is compiled, loaded with the data collection routines, and then executed in its normal test environment. During execution, probe data is collected while the program executes the test cases.

There are two levels of detail at which data is collected. When the desired measure of test effectiveness is DD-path coverage, path counts are collected in core and only the final values of the counts are written to a file which is input to the test analyzer. If sequence information on path executions is desired, a trace file may be written for input to the test analyzer.

The function of the test analyzer is to organize the execution data, generate reports, and accumulate the results of new tests into the test history. The basic reports are geared to a test objective of executing each DD-path at least once. The detailed reports show path

counts by path number and statement counts appended directly to the statements in the source listing. Summary reports provide a list of decision-to-decision paths not hit and measures of test effectiveness in terms of DD-path executions as a percentage of total DD-paths in a module. All reports are generated for single tests and test accumulation.

TESTING ASSISTANCE

The capabilities of the AVS described so far will carry a testing program through execution of existing test cases and provide DD-path coverage analysis of the results. Both JAVS and RXVP provide additional reports that are designed to assist the generation of ~~generate~~ new test cases and develop the understanding of the program organization that is essential to an efficient application of the human resources to the testing process.

A testing approach that makes use of these functions of an AVS is shown in Fig. 8. The data collection and analysis provides a measure of testing effectiveness in terms of coverage of DD-paths. A minimum testing goal is execution of all DD-paths in a program. This approach suggests minimizing human resources with automatic generation of test cases. At the simplest level the new test cases can be generated by some algorithm that systematically explores the input space. If a program runs without catastrophic failure, such a fully automated approach would serve well in an environment where a computer could be dedicated to exercising the program. If there were a few simple criteria for satisfactory performance, this system could also evaluate

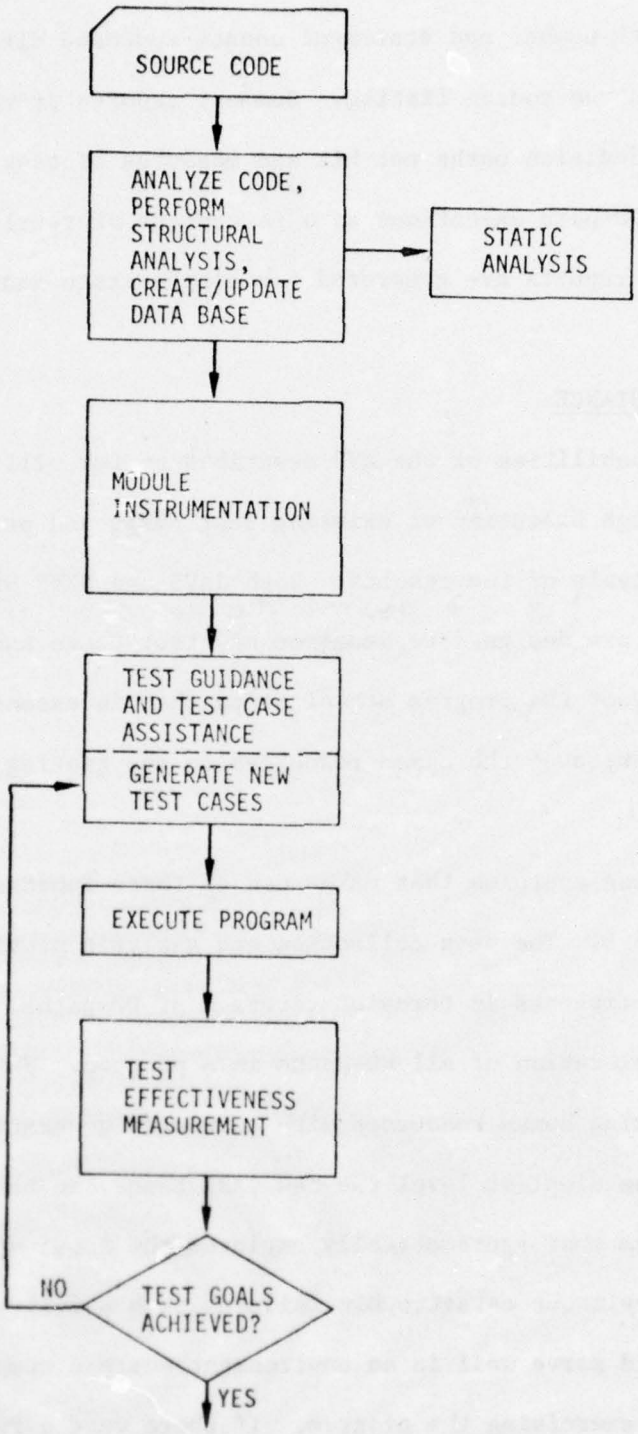


Figure 8. Program Testing with AVS

performance automatically. In our work with BMDATC, we are currently investigating this approach where the source of unacceptable performance is inputs that are beyond the capability of the system rather than mistakes in the implementation.

The above approach does not necessarily succeed in achieving full coverage of all DD-paths within some reasonable number of test cases and, of course, in the absence of a dedicated machine, this may be completely impossible because of the number of tests generated. One of the major shortcomings of this approach is that it does not provide much additional understanding of the implementation and what its internal structure is capable of. Another approach that uses fewer test cases, but which presently uses more human resource, is to direct the testing attention to particular untested DD-paths and attempt to generate test cases specifically for those paths. Figure 9 shows a more detailed look at the steps involved in that approach. The AVS provides direction for generating these test cases and ideally would produce them automatically.

The present versions of JAVS and RXVP are still dependent on human intervention to derive the input data needed to cause execution of a particular DD-path. This is greatly facilitated, however, through reports describing the decision outcomes and computations preceding entry to the desired DD-path.

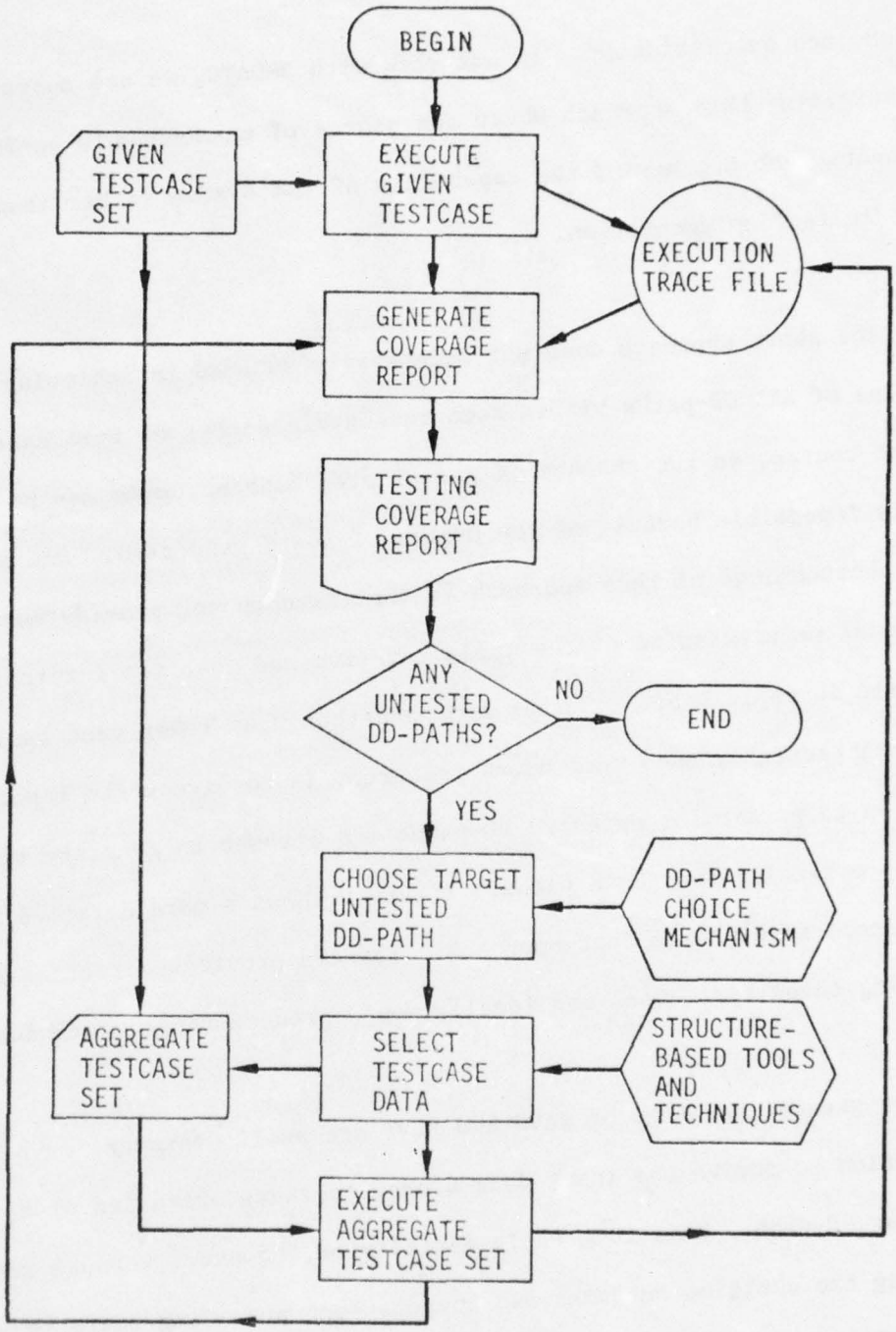


Figure 9. DD-Path Testing

The reports are the reaching set report (Fig. 10) and the sequence report (Fig. 11). The reaching set report gives the ur all DD-path sequences that provide connectivity between starting ending DD-paths. All predicates that have decision outcomes tha cause departure from the reaching set are flagged as essential a direct the tester to those conditions that must prevail in a tes to reach a particular testing target.

RXVP also provides the DD-path sequence report which produ detail on a particular sequence of DD-paths. Figure 11 shows th report, which includes a composite predicate for the path as wel tables of variables used and set along the sequence. Our experi version of RXVP also provides backtracking capability to simplif composite predicates and give more detailed information on how v set and used along the path affect the results.

Another capability that is supported in RXVP is conversion conventional FORTRAN into structured IFTRAN. This approach has esting possibilities for simplifying test case generation when a to reaching sets. Figure 12 shows a somewhat tortured segment o FORTRAN and Fig. 13 shows the same code rewritten in IFTRAN.

```
NON-ITERATIVE REACHING SET
   OF DD-PATH  17 FROM DD-PATH   4

MODULE <EXMPL1  >, JAVSTEXT <EXPROGM >, PARENT MODULE <EXPROGM >

-------------------------------------------------------------------------------------------------------------------------------
                 . . .
 18 ( 2)                 FOR J = 1 , 1 , LIMIT2 $
 19 ( 3)                   BEGIN
                 . . .
 21 ( 3)                   IFEITH J LQ 3 $                                  *ESSENTIAL PREDICATE*  BEGINNING DDP
                                                                          ** DD-PATH   4 IS TRUE BRANCH
 22 ( 4)                     INDXS = J $
                 . . .
 25 ( 3)                   END
                 . . .
 27 ( 3)                   SWITCH PICK = ( LABEL1 , LABEL1 , LABEL1 , LABEL2 ) $
 28 ( 3)                   GOTO PICK ($ INDXS - 1 $) $
                                                                          ** DD-PATH   7 IS SWITCH OUTWAY    1
                                                                          ** DD-PATH   8 IS SWITCH OUTWAY    2
                                                                          ** DD-PATH   9 IS SWITCH OUTWAY    3
                                                                          ** DD-PATH  10 IS SWITCH OUTWAY    4
                                                                          ** DD-PATH  11 IS SWITCH OUTWAY    5
 29 ( 3)           LABEL2.
                           GOTO EXMPL3 $
 30 ( 3)           LABEL1.
                           IFEITH RESULT LS 4 $
                                                                          ** DD-PATH  12 IS TRUE BRANCH
                                                                          ** DD-PATH  13 IS FALSE BRANCH
 31 ( 4)                     MESSAG = MSG2 $
 32 ( 3)                   ORIF RESULT EQ 4 $
                                                                          ** DD-PATH  14 IS TRUE BRANCH
                                                                          ** DD-PATH  15 IS FALSE BRANCH
 33 ( 4)                     MESSAG = MSG3 $
 34 ( 3)                   ORIF 1 $
                                                                          ** DD-PATH  16 IS TRUE BRANCH
 35 ( 4)                     MESSAG = MSG4 $
 36 ( 3)                   END
                 . . .
 38 ( 3)                   OUTPUT PRINTR MESSAG $
 39 ( 3)                   END                                              *ESSENTIAL PREDICATE*  ENDING DDP
                                                                          ** DD-PATH  17 IS LOOP ON FOR AGAIN
-------------------------------------------------------------------------------------------------------------------------------
```


Figure 10. Reaching Set Report

```
        1    2    5

DD-PATH LIST IN EXECUTION ORDER (NEGATIVE IMPLIES LEADER OF A PARALLEL DD

>    1 >    2 >   -5 >

COMPOSITE PREDICATE...

( .TRUE. ) .THEN. ( 1 .EQ. ISWTCH ) .THEN. ( .TRUE. .EQ. ( IFLAG .EQ. 0 )

STATEMENT TEXT(S) IN ORDER OF EXECUTION...
-------------------------------------------------------------------------
   [  1]          1            SUBROUTINE EXMPLE ( ISWTCH , IFLAG )
                  ------------------------------------------------------
   [  2]          2            GOTO ( 100 , 200 , 300 ) , ISWTCH
                  ------------------------------------------------------
   [  3,   4]     3      100   IF ( IFLAG .EQ. 0 ) PRINT 600 , ISWTCH
                               > > > > > TAKE THE .TRUE. BRANCH ↑ ↑ ↑ ↑ ↑
   [  5]          4            PRINT 601 , ISWTCH
   [  6]          5            GOTO 500
                               ... ... ...
   [ 22]         16      500   PRINT 605
   [ 23]         17            RETURN
                               ... ... ...
   [ 24]         24            END
-------------------------------------------------------------------------

THESE VARIABLES/CONSTANTS WERE            USED        ALONG THIS PATH...
      ISWTCH       IFLAG        0          600

THESE VARIABLES/CONSTANTS WERE            SET         ALONG THIS PATH...
      ISWTCH       IFLAG

THESE VARIABLES/CONSTANTS WERE IN PREDICATES          ALONG THIS PATH...
      ISWTCH       IFLAG        0

THESE VARIABLES/CONSTANTS WERE USED BUT NOT SET       ALONG THIS PATH...
      0            600

THESE VARIABLES/CONSTANTS WERE SET BUT NOT USED       ALONG THIS PATH...
      **NONE**

THESE VARIABLES ARE SET ALONG PATH AND ALSO USED IN SOME PATH PREDICATE.
      ISWTCH       IFLAG
```

Figure 11. DD-Path Sequence Report

```
      IF ( I .GT. IPTS ) GOTO 1500
      IF ( ABS ( XLONGA - XLONGB ) .EQ. 180. ) GOTO 90
      IF ( XLATA - XLATB ) 40 , 30 , 50
30    I = I + 1
      GOTO 10
40    JOUT ( I ) = 360
      GOTO 60
50    JOUT ( I ) = 180
60    DISTAN = ( XLXATA - XLATB ) * 60.
      IF ( DISTAN ) 70 , 80 , 80
70    DISTAN = - DISTAN
80    XLATA = XLATB
      XLONGA = XLONGB
      I = I + 1
      GOTO 10
90    IF ( XLATA .GE. 0 ) GOTO 96
      JOUT ( I ) = 0
      DISTAN = 180 - ABS ( XLATA )
      IF ( DISTAN - 180. ) 91 , 94 , 95
91    DISTAN = DISTAN * 60.
      GOTO 80
94    DISTAN = 180. * 60.
      GOTO 80
95    DISTAN = ( 360. - DISTAN ) * 60.
      GOTO 80
96    JOUT ( I ) = 360.
      DISTAN = 180. - ABS ( XLATA ) * 60.
      GOTO 80
10    CONTINUE
1500  CONTINUE
      RETURN
      END
```

Figure 12. Unstructured Reaching Set

```
IF ( I .LE. IPTS )
↓   IF ( ABS ( XLONGA - XLONGB ) .EQ. 180. )
↓   ↓   IF ( XLATA .GE. 0 )
↓   ↓   ↓   JOUT ( I ) = 360.
↓   ↓   ↓   DISTAN = 180. - ABS ( XLATA ) * 60.
↓   ↓   ELSE
↓   ↓   ↓   JOUT ( I ) = 0
↓   ↓   ↓   DISTAN = 180 - ABS ( XLATA )
↓   ↓   ↓   IF ( 0 .GT. ( DISTAN - 180. ) )
↓   ↓   ↓   ↓   DISTAN = DISTAN * 60.
↓   ↓   ↓   ORIF ( 0 .EQ. ( DISTAN - 180. ) )
↓   ↓   ↓   ↓   DISTAN = 180. * 60.
↓   ↓   ↓   ELSE
↓   ↓   ↓   ↓   DISTAN = ( 360. - DISTAN ) * 60.
↓   ↓   ↓   ENDIF
↓   ↓   ENDIF
↓   ↓   XLATA = XLATB
↓   ↓   XLONGA = XLONGB
↓   ↓   I = I + 1
↓   ELSE
↓   ↓   IF ( 0 .LT. ( XLATA - XLATB ) ) .OR. ( 0 .GT. (
↓   ↓   ↓   IF ( 0 .GT. ( XLATA - XLATB ) )
↓   ↓   ↓   ↓   JOUT ( I ) = 360
↓   ↓   ↓   ELSE
↓   ↓   ↓   ↓   JOUT ( I ) = 180
↓   ↓   ↓   ENDIF
↓   ↓   ↓   DISTAN = ( XLXATA - XLATB ) * 60.
↓   ↓   ↓   IF ( 0 .GT. ( DISTAN ) )
↓   ↓   ↓   ↓   DISTAN = - DISTAN
↓   ↓   ↓   ENDIF
↓   ↓   ↓   XLATA = XLATB
↓   ↓   ↓   XLONGA = XLONGB
↓   ↓   ↓   I = I + 1
↓   ↓   ELSE
↓   ↓   ↓   I = I + 1
↓   ↓   ENDIF
↓   ENDIF
ENDIF
```

Figure 13. Structured Reaching Set

FUTURE PLANS

We are continuing the investigation of test tools that are useful for large software system testing as described here. A primary source of experience to support those investigations will be the current testing being performed at RADC. The results of that effort will be reported in the early part of next year. In addition our experimental activities are addressing the following topics:

- Automatic documentation aids
- Test effectiveness measures
- Automatic test case generation
- Automatic structurizing
- Static analysis reports

ADVANCES IN TECHNOLOGY FOR VERIFICATION AND VALIDATION

Edward F. Miller

Science Applications, Inc.
211 Sutter St.
San Francisco, CA 94108

1. INTRODUCTION

The enormous amount of attention focused on issues of software reliability and effectiveness in the last decade has resulted from recognition of "software" as a potentially serious problem in computer-based systems. "Software" has taken the aura of the culprit, and remedial measures aimed at minimizing the chance that a serious error would escape unnoticed have demonstrated their effectiveness. Very loosely, these measures are called "verification and validation" (V&V) strategies.

At the outset, ad hoc procedures were used, with startling success. In the past five years, the body of techniques has become refined, and systematic application of them has become widespread. Almost simultaneously, and perhaps as a result of the need identified by the specific requirements of the technology base for V&V, research and development efforts began to address (and develop) a much stronger theoretical and methodological base.

The present state-of-the-art can be characterized as having reached a kind of plateau, both technologically and operationally. If the complexity of the software systems for which V&V is to be performed remained constant, there would be a (relatively) stable situation. The opposite is happening, of course: as software systems grow in overall complexity, the need for sophisticated methods for performing the V&V function grows accordingly.

The purpose of this paper is to assess the current state-of-the-art, and the prospects for future development, of three subtechnologies within the larger arena of V&V:

1. Techniques of systematic program testing as they can be applied de facto to large-scale (i.e., highly sophisticated) software systems.

2. Evolving methods of software design and production that offer the opportunity to produce bigger software systems more quickly and more reliably, and which possess features that make them amenable to a wide range of affirmation techniques.

3. Methods of requirements and specification analysis and decomposition that offer a secure bridge between system concept and system reality.

The discussion is partitioned into these areas because each represents a critical part of an overall "solution" to the software problem. The reasoning that supports this position is straightforward:

1. A well conceived statement of system requirements that is internally self-consistent forms the "standard" against which the final product (i.e., programs) can be compared.

2. A well-designed software system is intrinsically better suited to systematic testing as a means of verifying the consistency between the requirements and the final product.

3. Current and projected developments in program testing appear to offer the surest method of validating the relation between a software system's requirements and its final-product form.

Figure 1-1 illustrates the way in which these ideas interact to achieve confidence in a software system as the "solution" of the problem that was supposed to be solved.

Although the interactions illustrated in Fig. 1-1 may appear vague, they represent a feasible plan for achieving truly reliable software. As the technology-oriented material in the next sections will indicate, there is much to be done before this overall plan can become reality.

2. PROGRAM TESTING

Procedures for testing programs have undergone a minor revolution in the past five years; the revolution can be expected to continue. Techniques used divide into two natural categories:

a. Systematic de facto testing, which assumes that the software system being analyzed is stable.

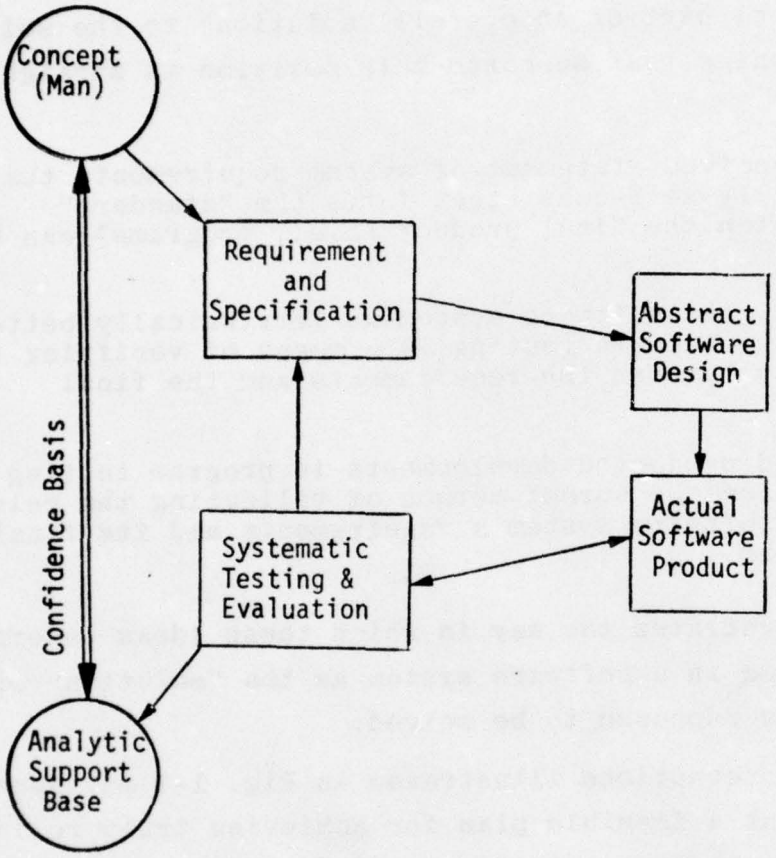


Fig. 1-1. Supportive Relationship Between System Concept and Analytic Basis

b. "Reliable" testing methods, which seek to affirm error-free status of parts of a software system by running specifically chosen tests.

Methodologies, and corresponding support tools, for the former area are well developed; the theoretical basis for the latter is only now being unraveled.

2.1 Structural Based Systematic Testing

Procedures for testing programs center on the development of a directed graph model of the program(s) (Ref. 1). This model is used to indicate a least-complex set of subprogram elements that are suitable targets for actual testing coverage measurement (Ref. 2). When every part of the program has been executed (and when that execution has been verified by independent means) one can be certain that no latent static faults reside within the software system.

A major problem in accomplishing this goal, both on a single-module (or unit) level and on a subsystem level, is the process of constructing new testcases that make the program (or program set) behave in a particular way. Manual, semi-automatic and automatic techniques (discussed later) have been devised to accomplish this.

Experience with the methodology associated with systematic program testing (Ref. 3) has, so far, indicated the general utility of the technique but has not achieved all of the results expected. As several people have pointed out (Ref. 4, for example), use of this approach is not without its successes. The major drawback appears to be the cost of implementing the process.

Higher levels of testing coverage, involving groups of modules and possibly taking into account the internal structure of the algorithms, have been suggested (Ref. 5), but are still in the process of being implemented.

2.2 Automation in Testing

The underlying theme of this species of formalized testing is "automation of function," which is done for two reasons:

1. To reduce the costs associated with applying a methodology to bearable levels.

2. To assure that the systematic testing process itself does not make any mistakes.

Progress in automating the functions has been substantial, as the brief review of major systems given next will serve to suggest.

The notion of a generalized program analysis facility was first embodied in the JAVS system, developed for RADC (Ref. 6). This system processes JOVIAL/J3 programs and provides basic functions of instrumentation, execution-history data collection, and machine-assisted support for generation of new testcase data. A related system, RXVP, uses the same techniques to process systems of Fortran programs (Ref. 7). A number of other systems--too numerous to mention specifically--are strong adherents to the basic philosophy established by JAVS. All of these automated tools, which might be considered the "second generation" of the genre, suffer primarily from high execution cost and lack of attention to the user interface. This is certainly not improper, since what is currently available tends to be in the form of "R&D prototypes." The body of experience gained from all of this work serves as a very strong basis for future development.

2.3 Integrated Systems

There are several ongoing activities aimed at producing machine-independent, language-dialect-independent, systems of automated program analysis tools. In one activity at NASA, for example, a full-spectrum approach is being applied in a way that culls the best methods from those available (Ref. 8) in other, earlier systems.

Figure 2-1 describes the major functions that such systems typically implement. In a current development activity (Ref. 9) a unified attempt is being made to develop the entire spectrum of program analysis functions on a non-proprietary basis. The intention of that activity is to have specific functions selectable by a user (a formal Test and Evaluation activity), and provide a system and

ATA Automated Testing Analyzer. Provides the basic "window" into the program testing process...tells the percentage coverage in terms of a very simple (yet powerful) measure.

DAP Dynamic Assertion Processor. The programmer puts assertions about the way programs are supposed to behave into the code (they're treated as comments by the compiler), but the tool makes them report on when the assertions are violated.

SMI Self-Metering Instrumentation. The programs are completely instrumented (but logical integrity is preserved) so that all conceivably interesting data about each programs' execution-time behavior is collected and reported (under user command).

TDE Testing Difficulty Estimator. Tells, according to detailed program structure analysis, which programs in a set are the easiest and most difficult to test...and helps management control the testing resource better.

ATG Automated Testcase Guidance. Helps a user construct test data that meets easily-stated objectives (...execute this path...).

ATDG Automated Test Data Generation. Automatically generates test data that meets specific objectives, using advanced heuristic processes that have high success ratios for this very difficult problem.

ATF Automated Test Facility. Constructs a stubbed and standard Input/Output environment for a single program or a set of programs...all automatically.

SAA Static Allegation Analyzer. Applies advanced techniques to analyze programs statically to prove important allegations about the programs' behavior dynamically...for example, that "all variables are set before they are used."

RTA Reliable Test Analzyer. Checks out the reliability of a test case in protecting against assignment and/or control errors...thereby helping to maximize the sureity attained in a rigorous testing activity.

RLP Robust Language Processor. Automatically rewrites a program so that it "can't fail" during execution without first telling the user exactly how it did fail. Used during debugging and checkout testing to (a) isolate mistakes and (b) minimize lost execution time.

TPS Test Planning/Status. A tool for testing management that provides continual status checks on the testing activity, and an archive for test data.

AMA Automated Modification Analyzer. Automatically analyzes a program set for the potential impact a proposed program change will have on the testing process...and advises what re-testing will have to be performed as a consequence of the change.

Fig. 2-1. A typical integrated family of test and evaluation tools.

language-specific "package" tailored to the user's actual
Significant attention is being paid to the "user interface
well as the volume of analytic output produced.

2.4 Automatic Testcase Generation

The issue of generating new testcase data has receive
attention recently; achieving this goal seems to lie at th
of widespread application of systematic, automated testing
ogies. There have been many notable efforts, but only a f
The general problem of testdata generation is known to be
undecidable (Ref. 10); consequently, any technique used m
heuristics of some kind.

A system of refining new testcase data from known inf
about existing testcases appears to be the most effective
(Ref. 11). This system adapts a testcase for which the ex
behavior is already known to a new purpose--presumably to
testing coverage of some as yet untested program element.
istics required appear to be achievable in practice for pr
programming languages such as FORTRAN and COBOL.

Related techniques of symbolic execution (Ref. 12,13)
promise so long as the user is interacting with the proces
alyzing potential program flows. Symbolic execution techn
vide the manipulative basis for dealing with the complex s
of "real" programs with acceptable execution costs. In on
effort the focus is on symbolic evaluation methods that pe
veloping testing coverage data for assembly language progr
reference to actual execution at all (Ref. 14).

2.5 Reliable Testing Theory

In a landmark paper by Goodenough and Gerhart (Ref. 1
shown that reliable methods of program testing can be equi
a formal program proof of correctness. So long as certain
are met, it appears that it is possible to devise tests th
"proof" against many classes of program errors. Howden (R

has assessed the utility of the approach and found (in in terms) that in fact this is the case except for programs structural faults. Thus, the theory suggests that it sho sible to employ testing as a means for certifying program so long as potential program errors (within the classes c do not affect the structural properties of the algorithm plemented.

The reverse of this argument may offer the solution ing against structural flaws: if structural-based reliab are effective against non-structural errors, then non-stru testing may be effective against structural areas (Ref. 2

If this notion is fully elaborated, the significance enormous because it would imply the existence of an integ methodology for satisfactorily testing out faults in a ca software system, regardless of whether they are structural structural. An important area for investigation is the a of automated test generation techniques to the production tests only.

2.6 Summary

An evidently continuous spectrum of methodologies, a quite continuous span of associated automated tools, appe evolving. The ultimate effect will be to systematize the verifying that an actual software system indeed satisfies tional requirements imposed on it. The "bridge" between gram tedt and requirements text appears in the context of to partition programs' input spaces according to stated r (see below). Once this is accomplished, establishing the necessary to affirm a system designer's reliance on the v the software would be reduced to a human (and, potentiall assisted) understanding problem.

3. SOFTWARE DESIGN/PRODUCTION

The second ingredient in the "formula" suggested in Sec. 1 includes the use of advanced techniques of software design and production. It is necessary to distinguish between software and requirements/specifications because they deal with different entities: software design involves the incremental refinement of a statement of the form and content of a set of programs, whereas requirements/specification analysis must be performed in terms of abstractions of (human or machine) understanding of the problem being solved.

Advances in the realm of machine-assisted software design are only now being reduced to practice, following a period of consolidation within the software community that is focusing on two disciplines: structured programming and hierarchical design.

3.1 Structured Programming

The computer science community's love affair with Structured Programming probably owes its existence to requirements in the late 60's for a more widespread technology of programming to apply to then-current development projects. The basic notions of structured programming are simple, effective, and contagious.

The actual focus of research and development, has, however, been broader than simply providing a means to better deal with programs proper. As the RADC Structured Programming Series (Ref. 16) suggests, much more is involved. It can be assumed that the body of information contained in that Series will become the standard interpretation of the meaning of Structured Programming.

An important part of the new-style process for developing software designs is the use of HIPO (Hierarchical Input/Process/Output) charts as part of the supporting media (Ref. 17,18). Combined, these mechanisms provide the framework for a systematic approach to designing software.

3.2 Automation in Software Design

The "translation" of a requirement (or specification) into the design for a working peice of software is an art, at least as it is currently practiced. There is reason to believe that there never can be a "science" that describes the design process; this does not mean that the media used, and the transformations between them, cannot be treated scientifically. The actual conceptualization of an appropriate algorithm is an abstract process not amenable to rigorous treatment.

This, if one's purpose is to minimize the cost of software design the route by which this can be done is to automate the processing of the media. This philosophy meshes well with the facts (as they are understood): human input is necessary to achieving good software design.

Automated Software Production systems, such as are currently being developed (Ref. 9), attempt to (a) minimize the low-level effort required to capture a good software design, and (b) maximize the usefulness of every input provided by the human. For example, current concepts suggest that the overwhelming majority of detailed cross-referencing and other low-level tasks--those that can be performed by machine--should be automated. At the same time, a generalized database supporting the software design process assures that there is a continuous archive of the entire process. Interactive use by skilled software designers is assumed. Much of what currently is acceptable as program documentation is produced automatically, after the design process is complete.

3.3 Automation in Software Production

Present software production processes still involve significant manual input; there is still a need for programmers to "debug" programs. (This is an activity distinguished from formal testing by the fluid state of the program text at that point.) Automating the

actual production process has been the subject of numerous papers and reports. Some of the best ideas seem to be:

1. The use of a generalized macroexpansion capability to minimize the need for repeating often-performed actions.

2. The development of sophisticated conditional editing functions, which would allow generalized programming independent of specific machine environments and would automate the process of producing specific program texts for differing environments.

3. Self-hosting of the software production system (so that it is used to produce a working copy of itself), and the use of "portable programming" methods to assure intrinsic transportability. FORTRAN and COBOL subsets are envisioned as practical universal-implementation language bases.

4. REQUIREMENTS/SPECIFICATION DECOMPOSITION

The paramount problem facing the V&V community is the development of a rigorous method for evolving a consistent set of system requirements (or system specifications), particularly as they apply to the embedded computer element. Requirements/specifications differ from a software design in that they address the actual real-world problem; they are similar in that iterative decomposition techniques, combined with automated consistency analysis, can be used to develop them.

There have been few actual experiments involving the use of hierarchical decomposition techniques in developing specifications (or requirements) for computer software systems. What has been done is promising. For example, the tools provided by ISDOS (Ref. 19) have been used in this fashion.

What seems to be lacking in this area is (i) a sufficient level of machine-based transaction processing (each new intellectual input represents a single transaction), and (ii) the basis for developing internal consistency checking.

4.1 Consistency Checking

The cross-links that are needed for automatic consistency checking can be inserted manually, provided that a syntax appropriate to the task is provided. If this is accomplished, the database processing needed becomes relatively trivial. One project (Ref. 20) appears on the verge of accomplishing this.

The outcomes of an automated consistency check procedure are less powerful than one might imagine, but hardly powerless. Although it is impossible to prove the internal consistency of a set of requirements/specifications (Ref. 19), the capability to identify many classes of inconsistencies serves to strengthen the interpretable validity of the document.

4.2 Natural Language Analysis

The other method is to employ full- or restricted-English in concert with a natural language processing system (Ref. 21). In this case, one would transact primarily in English (or near-English) and the associated language processor would develop the links needed for the consistency checking. Current advances in the techniques of parsing English, and developing alternative semantic tree interpretations of the meaning of English sentences, have the promise of being highly effective in this particular role.

Current research appears to be producing a number of highly automated systems which implement good (non-trivial) algorithms, and which provide significant relief from the logistics of developing, analyzing, and archiving an embryonic requirement/specification document. One would hope that techniques of program decomposition (Ref. 5) can be extended to stated requirements, so that a body of methodologies for analyzing requirements--much in the same way as programs are now analyzed--can be used to resolve ambiguities effectively.

REFERENCES

1. M. R. Paige, "Program Graphs, An Algebra, and Their Implication for Programming," IEEE Trans. Software Engineering, Vol. SE-1, No. 3, September 1975.

2. E. F. Miller and M. R. Paige, "Concepts of Automated Testing Analysis," Science Applications, Inc., Software Technology Center, RP-1, September 1975.

3. E. F. Miller, "Methodology for Comprehensive Software Testing," AD-A013-111/OWC, September 1974.

4. J. Brown, "Why Tools?", Proceedings of Workshop 4, Computer Science and Statistics: Eights Annual Symposium at the Interface, UCLA, 1975.

5. E. F. Miller, Jr., "The Art and Theory of Program Testing," Proceedings of the Conference on Computing Systems, Austin, Texas, 1975.

6. "JOVIAL Automated Verification System (JAVS)," RADC TR-76-20, 1976.

7. "RXVP: An Automated Verification System for Fortran, Reference Manual," General Research Corporation, Santa Barbara, CA, 1975.

8. J. P. Ryan, "A System for Automatic Software Evaluation," Proc. 1976 Summer Computer Simulation Conference, Washington, D.C., 1976.

9. In progress at Software Technology Center, Science Applications, Inc., 211 Sutter St., San Francisco, CA 94108.

10. W. H. Howden, "Reliability of the Path Analysis Testing Strategy," University of California, La Jolla, CA, June 1975.

11. M. A. Holthouse, "A Tractical System for Automatic Testcase Generation," Presented at 1976 National Computer Conference, New York City, 1976.

12. J. King, "Symbolic Execution and Program Testing," Presented at 1976 National Computer Conference, New York City, 1976.

13. W. H. Howden, "Experiments with a Symbolic Evaluation System," Presented at 1976 National Computer Conference, New York City, 1976.

14. K. Heffner, et. al., "Final Report: Symbolic Evaluation/ Execution Study," Science Applications, Inc., Software Technology Center, November 1976.

15. J. B. Goodenough and S. Gerhart, "Toward a Theory of Test Data Selection," 1975 International Conference on Reliable Software, Los Angeles, CA, 1975.

16. Structured Programming Series, RADC TR-74-300 (15 volumes), 1975.

17. J. F. Stay, "HIPO and Integrated Program Design," IBM Systems Journal, 1976.

18. E. Yourden, Techniques of Program Structure and Design, Prentice-Hall, 1975.

19. D. Teichroew, "PSL/PSA: A Computer Aided Technique for Structured Analysis and Design of Information Processing Systems," University of Michigan, 1975.

20. B. Buckles, et. al., "NASA Software Specification and Evaluation System Design," NASA, Marshall Space Flight Center, 1976.

21. S. R. Petrick, "On Natural Language Based Computer Systems," IBM J. Research and Development, Vol. 20, No. 4, July 1976.

22. M. R. Paige, "Toward Database Verification," Proceedings of Eurocomp-1976, England, 1976.

Advances in Technology for Verification and Validation

Edward F. Miller, Jr.

OBJECTIVES OF THIS TALK

- ASSESSMENT OF CURRENT PRACTICE
- EVALUATION OF CURRENT DEVELOPMENT TRENDS
- INDICATION OF FUTURE GROWTH DIRECTIONS

DEFINITIONS

VERIFICATION: THE PROCESS OF ASSURING THAT WHAT IS INTENDED TO BE PRESENT IN SOFTWARE IS ACTUALLY THERE.

VALIDATION: THE PROCESS OF ASSURING THAT WHAT IS ACTUALLY PRESENT IN THE SOFTWARE SOLVES THE RIGHT PROBLEM.

THREE PRIMARY AREAS WITHIN THE LARGER FRAMEWORK OF VERIFICATION AND VALIDATION:

- SYSTEMATIC PROGRAM TESTING
- AUTOMATION IN SOFTWARE DESIGN/PRODUCTION
- AUTOMATION IN REQUIREMENTS/SPECIFICATION DECOMPOSITION

MEASURING TESTING "COVERAGE"

OBJECTIVE:

QUANTIFY HOW WELL A SET OF TESTS AFFIRMS THE "QUALITY OF A PROGRAM

CONSIDERATIONS:

- WHATEVER IS USED, IT HAS TO APPLY TO LARGE SOFTWARE SYSTEMS
- THE SECOND INSTANCE OF A TEST SHOULD NOT INCREASE THE "RELIABILITY" VALUE
- EVERYTHING IN THE PROGRAM HAS TO BE TESTED
- ALL THE THINGS NOT IN THE PROGRAM HAVE TO BE TESTED, TOO
- THE TESTS HAVE TO BE RELIABLE, IN AGGREGATE

PROGRAM STRUCTURE

HIERARCHY OF TESTING COVERAGE MEASURES

(CO) PROGRAMMER'S INTUITION

(C1) EVERY STATEMENT EXERCISED AT LEAST ONCE

(C2) EVERY PROGRAM PREDICATE OUTCOME ACHIEVED AT LEAST ONCE

(C3) AT LEAST ONE ELEMENT OF EACH EQUIVALENT CLASS OF PROGRAM FLOW EXERCISED AT LEAST ONCE

(C4) ALL USEFULLY DISTINCT PROGRAM FLOW CLASSES, TESTED WITH A "RELIABLE TEST," PLUS DE FACTO TESTING OF WHAT CANNOT BE TESTED RELIABLY

(C_n) A SUFFICIENT SET OF CLASSES OF PROGRAM ACTIVITY TO PROVIDE TESTS WHICH PROVE THE APPROPRIATENESS OF THE PROGRAM TO THE NEEDS IT IS SUPPOSED TO FILL

RELIABLE TESTING THEORY

BASIC REFERENCES: GOODENOUGH AND GERHART, "TOWARD A THEORY OF TEST DATA SELECTION"

HOWDEN, "RELIABILITY OF THE PATH ANALYSIS TESTING STRATEGY"

AD-A033 625 ROME AIR DEVELOPMENT CENTER GRIFFISS AFB N Y F/G 9/2
PRESENTATIONS AT THE RADC/AR?A INVITATIONAL DOD/INDUSTRY CONFER--ETC(U)
AUG 76

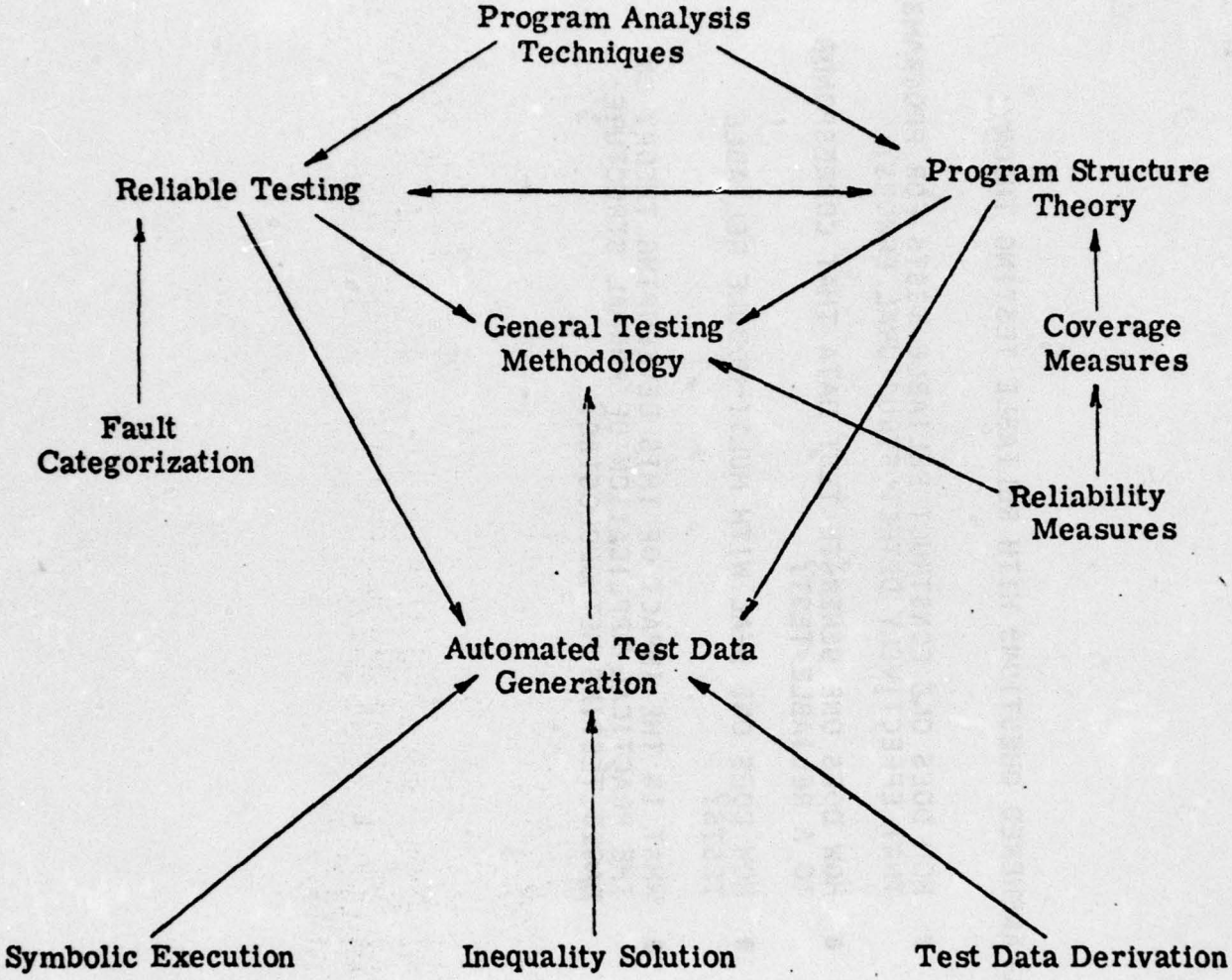
UNCLASSIFIED NL

3 OF 5
AD
A033625



A PARAPHRASED STATEMENT OF A BASIC RESULT ON RELIABLE TESTING THEORY:

"UNDER CERTAIN CONDITIONS IT IS POSSIBLE TO CONSTRUCT TESTS OF PROGRAMS SUCH THAT NON-STRUCTURAL ERRORS CAN BE DISCOVERED WHEN THE TESTS ARE EXECUTED"


Program Analysis Techniques
Reliable Testing
Program Structure Theory
General Testing Methodology
Coverage Measures
Fault Categorization
Reliability Measures
Automated Test Data Generation
Symbolic Execution
Inequality Solution
Test Data Derivation

UNANSWERED QUESTIONS WITH RELIABLE TESTING THEORY:

- HOW DOES ONE CONSTRUCT RELIABLE TESTS FOR PROGRAMS THAT EFFECTIVELY DETECT STRUCTURAL ERRORS?
- HOW DOES ONE GENERATE TEST DATA THAT CORRESPONDS TO A RELIABLE TEST?
- HOW DOES ONE DEAL WITH MULTI-MODULE RELIABLE TESTS?
- WHAT IS THE IMPACT OF THIS DEVELOPING THEORY ON THE PRACTICAL APPLICATION OF FORMAL STRUCTURE-BASED TESTING METHODOLOGIES?

BASIC CHARACTERISTICS OF GENERATIONS OF PROGRAM ANALYZERS

FIRST GENERATION: Simple Tools for Feasibility Demonstrations of Single Functions

SECOND GENERATION: More Complex Functions Performed

Partially Integrated

Experimental Prototypes

THIRD GENERATION: Production Tools:

- o User Engineered
- o Proven Algorithm Design Techniques
- o Efficient Implementations
- o Multiple Language/Multiple System Installation (Portability)

FIRST GENERATION PROGRAM ANALYSIS TOOLS

- OPERATIONAL FOR FORTRAN (EXCEPT AS NOTED)
- TYPICAL EXAMPLES:

  TDEM
  NODAL
  SOFTOOL /RSUP
  FETE
  FORTUNE
  COTUNE (COBOL)
  ACES

- LIMITATIONS:
  - o Incomplete Algorithms In Many Cases
  - o Costly To Use
  - o Poor User Interface
  - o Non-Portable

SECOND GENERATION PROGRAM ANALYSIS TOOLS

- ORIENTED TOWARD FORTRAN AND SOME OF ITS DIALECTS
- TYPICAL EXAMPLES:
  - PET
  - RXVP
  - SELECT
  - DAVE
  - PACE
  - JAVS (JOVIAL/J3)
  - FACES
  - AMPIC
- LIMITATIONS:
  - Excessively Cumbersome
  - Voluminous Output (Overkill)
  - Weak User Engineering
  - Non-Portable
  - Difficult To Extend To Other Dialects

THIRD GENERATION PROGRAM ANALYSIS TOOLS

- OPERATIONAL FOR MOST PROCEDURE ORIENTED LANGUAGES, PLUS ASSEMBLY LANGUAGE (WITH APPROPRIATE RESTRICTIONS)
- COMPLETELY PORTABLE, MACHINE INDEPENDENT
- USER-ENGINEERED
- COMMON DESIGN BASIS
- HIGHLY EXTENSIBLE
- PROVEN DESIGN TECHNIQUES
- INTEGRATED MULTI-LANGUAGE/MULTI-SYSTEM APPROACH
- MINIMAL EXECUTION TIME/SPACE REQUIREMENTS

AUTOMATED SOFTWARE DESIGN/PRODUCTION: SOFTWARE PRODUCTION METHODOLOGY

BASIC INGREDIENTS:

- GENERALIZED DATABASE
  - CURRENT DESIGN/SOFTWARE TEXTS
  - ARCHIVE
  - CONFIGURATION CONTROL
- MACROEXPANSION
  - MULTI-LEVEL MACROS
  - SELF-IMPLEMENTATION MACROS
- CONDITIONAL EDITING FEATURES
  - USER PARAMETER SETTINGS
  - QUERY RESOLUTION (INTERACTIVE MODE)
- PRODUCTION CONTROL/STATUS

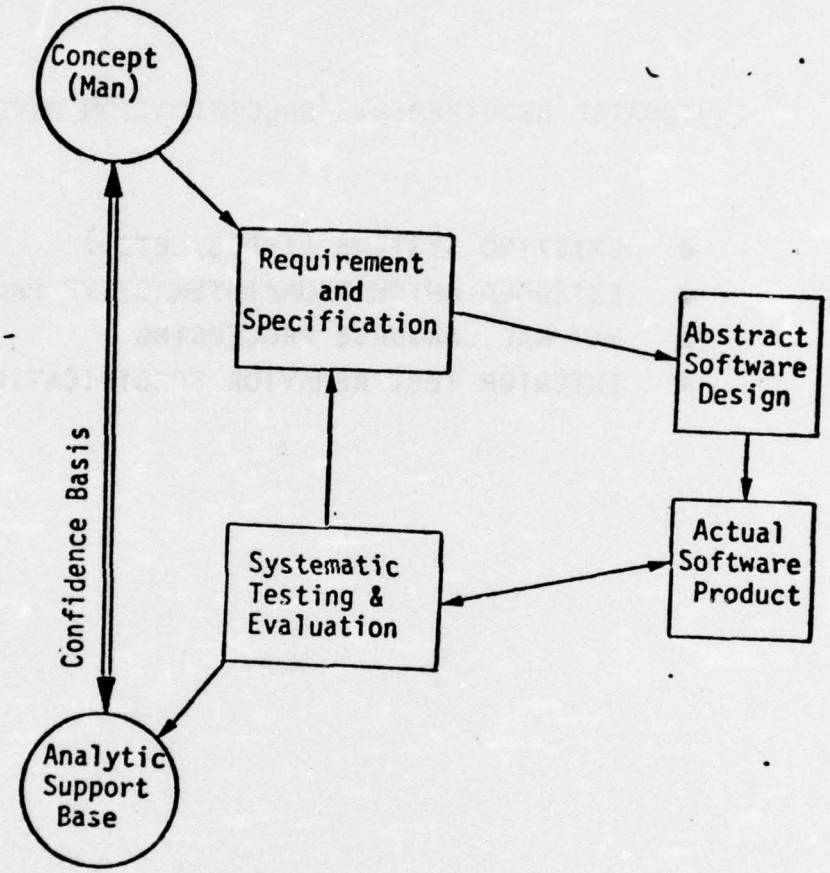
AUTOMATED SOFTWARE DESIGN/PRODUCTION: SOFTWARE DESIGN METHODOLOGY

BASIC INGREDIENTS:

- HIERARCHICAL DECOMPOSITION
  - PROGRAM STRUCTURE
  - DATA STRUCTURE
- PSEUDOCODING
  - ALGORITHM SELECTION
  - ALGORITHM STRUCTURE
- STRUCTURED PROGRAMMING
  - BASIC CONSTRUCTS
  - EXTENDED CONSTRUCTS
- AUTOMATION FEATURES
  - CROSS-REFERENCING
    - MANAGEMENT REPORTING

AUTOMATED REQUIREMENTS/SPECIFICATION DEVELOPMENT

- EXISTING SYSTEMS (ISDOS, ETC.)
- EXTENDED AUTOMATION/INTERACTIVE PROCESSING
- NATURAL LANGUAGE PROCESSING
- EXTERIOR TEST BEHAVIOR SPECIFICATION


Concept
(Man)
Requirement
and
Specification
Abstract
Software
Design
Actual
Software
Product
Systematic
Testing &
Evaluation
Analytic
Support
Base
Confidence Basis

THE IMPACT OF VERIFICATION AND SECURITY GOALS ON PROGRAMMING LANGUAGE DESIGN

by

Ann E. Marmor-Squires

Department of Defense

Fort George G. Meade, Maryland 20755

ABSTRACT

In developing software to perform highly critical control functions or to process classified or sensitive information, the ability to demonstrate that the software is performing its specified function correctly and that the integrity of the data is maintained is of vital concern. This paper shows how the goals of program verification and data security have influenced the programming language to be used for software development. It is suggested that certain programming language concepts and mechanisms (namely, abstraction, modularity and access control) facilitate achieving these goals and impact the design of the language itself. Abstract data types are suggested as an appropriate vehicle for their realization in a programming language. This thesis is supported by examples of several recent languages incorporating the notion of abstract data types that have been designed to support software engineering techniques and to facilitate verification.

KEY WORDS

abstract data type, abstraction, access control, programming methodology, programming language, protection, reliability, security, verification.

INTRODUCTION

The goal of developing software that satisfies rigorous security requirements and that can be formally verified to always execute in accordance with precisely stated specifications of its behavior is ambitious. If at the same time the software is not be to grossly inefficient or highly costly it is quite ambitious. Although absolute guarantees are unlikely, a degree of confidence significantly better than has

been traditionally provided that the software performs as desired is possible. Obviously, a critical feature of the software is the programming methodology and the programming language used to develop it. This paper identifies several principles and concepts playing a key role in programming language design that hold promise of helping achieve correct and secure software. The interaction of these principles and concepts and their impact on the software and the language in which it is written is presented in this paper.

SECURITY APPLICATIONS

It has become generally recognized that designing and implementing software and then testing it for an arbitrary subset of possible inputs to detect any errors is inadequate to demonstrate that the program meets its specifications and is error-free. When a subset of these specifications describe the security requirements that the software is to meet, it is clear that this traditional method of development is also inadequate to demonstrate that the software is secure. Security used in this context is a subset of the total system reliability: it refers to that portion of the software's adherence to specifications dealing with the flow of privileged information. Naturally, security is heavily dependent on the environment in which the software will operate and on the application for which it will be used. For the purposes of this discussion, a secure program will be defined as one which

(i) performs what is intended

(ii) performs nothing else

(iii) protects itself and the data it is handling from unauthorized modification, disclosure or destruction,

The adherence of a program to (ii) and (iii) differentiates a secure program from a reliable program and is of vital concern in applications involving the processing and controlling of classified or sensitive information.

We now briefly describe two typical security applications. Consider an application which processes sensitive data, e.g., a file containing military classified data or "Privileged Personnel Information". For various reasons, it is desirable to have only one file with various processes operating on it and sharing the data contained in it. It is also desirable not to permit uniform access to the file; control and enforcement of selective access is often more appropriate.

A second application is software which is controlling a security function, e.g., message segregation and routing. In this case, a case,a decision is made and subsecuent action is taken based on data in the message. Assuring that the integrity

of the data is maintained, that the security function is operating correctly and that it cannot be bypassed are of critical concern. These applications will help focus the discussion and should provide a useful basis for considering some of the ideas developed later in the paper.

FOCUS ON PROGRAMMING LANGUAGE

Major obstacles in achieving reliable and secure software have been the difficulties involved in stating precise formal specifications for what the software is to do, formally defining what security of the software means and then formally demonstrating that these specifications have been met. Often security requirements have been nebulous and left to implicit understanding resulting in unanticipated or unacceptable software behavior. For the typical applications briefly defined earlier a formal definition of system security specifications appears to be a doable task; however, significantly better support from the programming methodology and the programming language for software development is crucial.

Attention is being focused on programming languages for the following reasons:

* A programming language is the vehicle for man/machine communication used to evoke computations. There is some evidence that the design of the language itself determines how the computation to be performed will be expressed. Using a language well-suited to the problem permits better communication of the functionality of the computation free of needless detail. This is highly desirable for security applications for which verification of correct and secure operation is deemed important.

* It is becoming clear that the language chosen for developing software is vital if we intend to say something about that software with a high degree of confidence. Certain language features are error-prone constructs; other language features are known to make program verification difficult. It has also been claimed that the ability to verify programs not written with the intention of being verified or to verify programs written in "classical" languages not designed with ease of verification in mind (without suitable restrictions) are unreasonable goals for program verification [Fisher 75].

* If the verification of program properties (such as correctness and security) is to become a practical reality, significantly better language support is necessary to keep the task manageable. In particular more adequate support for incremental design, implementation and proof and for program structuring and modularizing into small independent pieces such

that the proof may be carried out on a module-by-module basis are some of the key issues.

Several software concepts have been identified that can facilitate the formal stating and verifying of security-related properties and thus help achieve the goal of reliable and secure software. These key concepts are abstraction, modularity and access control. The next section briefly defines them and discusses their influence on the design of programming languages to support verification and security.

ABSTRACTION, MODULARITY AND ACCESS CONTROL

The principle of abstraction is a key concept in controlling complexity in software development. It refers to the process of identifying essential properties that are common to entities and neglecting their inessential details. This definition of abstraction implicity includes the notion of information hiding advocated by Parnas [Parnas 72b] which highlights the importance of making inessential information inaccessible. The idea of using abstractions in developing software is not really new. High level programming languages provide some degree of abstraction in the sense that one need not program at the assembly or machine language level and be concerned with details at that level. Many high level languages also provide another effective abstraction mechanism, namely, routines or procedures, for grouping a sequence of language statements logically to perform a specific function.

The abstraction mechanism found in conventional languages enabling a programmer to write procedures or functions, refer to them by name and use them in other portions of his program does not provide a sufficiently rich set of abstractions. Functions and procedures correspond to what has been termed "abstract operations"; however, they operate on "concrete data". Procedures perform operations on data that are tied to the representation of that data and are often meaningful only with respect to that particular implementation. If the data structure implementation is modified, all the procedures operating upon it frecuently will require modification to deal with the new implementation. If a language abstraction facility were available which expresses the relationship between these abstract operations and the abstract data structure (e.g., stack, list, binary tree) on which they operate, the need for changing procedures when data implementations (concrete data) are changed could be essentially eliminated. Unfortunately, languages today do not adecuately provide for the expression of this relationship.

In addition to ease of program modification, other benefits could ensue from better language support for abstraction. By associating abstract operations with the corresponding data structure, it should be easier to determine whether the abstraction is complete, i.e., whether all the essential features of the concept being abstracted are explicity covered and the concept adequately characterized. Detection of inconsistencies or inaccuracies should be facilitated since the unimportant detail is suppressed and the important characteristics explicity expressed.

The abstraction facilities provided by the language greatly influences the ease of program verification. It should be easier to verify properties of the abstraction independent of its concrete realization since there is a well-defined set of abstract operations manipulating the abstract data structure. In particular, the verification of properties related to the integrity of the data should be facilitated because "there are less things to look at and consider" dealing with the data at that level. It should be possible to verify that invariant properties of a data structure (e.g., the mathematical characterization of a binary tree or a property of stacks that pushing an element onto the top of the stack followed by popping the top element makes visible the element on top of the stack just prior to the push) are maintained independent of what particular implementation is chosen. Once the properties of the abstraction are established and verified, the program verification task consists of demonstrating that the concrete representation correctly implements the abstraction. There is growing evidence that this two level process makes practical verification more manageable [Wulf 76, Ambler 76b].

Better language support for abstraction should also ease the programming task in that inessential implementation detail can be ignored during the initial phases of software development. It also would encourage concentrating on the problem solution in terms relevant to the problem domain. Of course, implementation issues will have to be dealt with; however, allowing those decisions to be deferred until the appropriate time should ease the programming task and keep it manageable.

Abstraction provides little value for practical applications without being used in conjunction with modularity. Essential to the use of abstraction is the assurance that appropriate abstractions are being utilized. The concept of modularity in software development can help guide this process. Research on modularization techniques has recently focused on methods of dividing large programs into small independent modules in such a way that they interact only in predictable and precisely specified ways [Wirth 71, Parnas 72a]. The issue of concern in modularization is not in keeping modules small by any measure as much as it is a question of module localization, non-interference and well-defined interaction. Many of these ideas are captured in the modular decomposition process advocated by Parnas [Parnas 72b] in which a program is developed

as a hierarchical structure of units (modules), each unit being the realization of some abstraction and further elaborating concepts introduced at a higher level. The idea of "hiding" is further advocated by Parnas as the major criterion for the decomposition of the program into appropriate modules. Hiding makes visible (external to the module) only those properties necessary to interface with other modules. In particular, the caller of a module is restricted to precisely this external interface information and not to the details of the module implementation. This explicit statement of properties necessary to users of a module (including when a module call will evoke an error condition) should facilitate the verification that the module interaction is as specified. This is a key step in verification of security properties of a program.

The concept of access control, a term drawn from research in operating systems, is the third key idea in achieving secure programs. Access control mechanisms are used in operating systems to control sharing of data and to protect against unauthorized access to data. They explicity define and enforce constraints on the sharing of data. Current programming languages do not provide adecuate support for selective and controlled sharing of data. Consider the typical security applications whose characteristics were briefly defined earlier in the paper. Uniform access to a file by all the program units which are to manipulate the file is often highly undesirable. Assuring that the integrity of some data is maintained and that programs are well-behaved (exhibit behavior according to its specification and none other) is also of concern. For such applications, incorporation of an access control mechanism into a programming language to explicitly allow controlled and selective access to data should help facilitate the verification of properties needed to demonstrate a program's secure behavior. A recent paper [Jones 76] illustrates how an access control facility can be incorporated into a class of programming languages that the author's call "object-oriented" languages. Such languages allow the definition of abstract data types: data is viewed as an abstract object and associated set of abstract operations for manipulating the object. Benefits gained from such a facility are described in that paper.

The notion of abstract data types is being suggested to tie all of these principles together and realize them in a programming language. We feel that abstract data types are an appropriate vehicle for expressing abstractions, an appropriate unit of modularity and an appropriate vehicle for embedding an access control mechanism in a programming language. The next section defines what is meant by an abstract data type.

ABSTACT, DATA TYPES AND THEIR IMPLEMENTATION

An abstract data type can be thought of as an extension and modification of the concept of data type. It is "abstract" and differs from the common notion of a data type in that an

abstract data type is used at one level but realized at a lower level. The definition of an abstract data type includes:

(i) definition of the operations applicable to objects of that type

(ii) a declaration of a representation for abstract data objects

(iii) realization of the operations on the representation.

As a very simple example, consider the abstract data type "stack". A possible set of operations that are meaningful to objects of type stack consist of

push	pushes an element onto the top of the stack unless full
pop	pops the top element off the stack and does nothing if the stack is empty
top	returns the value of the top element of the stack
empty	returns the value TRUE if and only if the stack is empty

(These definitions would be stated more formally for use in verifying the correctness of (iii) and as the visible specification for stack. They are stated informally here in order to illustrate the concept.)

This set of operations available on type stack completely defines the characteristics of its behavior and captures the abstraction of stack. These operations are meaningful for the type stack, not for the concrete representation of stack. The second portion of the stack definition is a declaration of a representation for objects of type stack. A typical choice is the representation as elements of an array. Note that the user of an abstract type stack need not know that the concrete representation of a stack is an array. The user of the abstract type should be unaffected if the representation is changed to a list, as long as it conforms to the stated specifications of the stack operations. (We are assuming a proper implementation, of course.)

The concept of abstract data types was first implemented in the class construct in Simula 67 [Dahl 68]. Simula classes were designed to represent and permit unrestricted accessibility to data objects. Class attributes and functions are accessible in the block in which the class definition is embedded; thus the actual form of the data representation can be accessed. Although protection was not a feature of Simula 67, several protection mechanisms have been suggested as language extensions [Hoare 72, Paine 73, Spitzen 76].

The current notion of abstract data type includes the protection of the data representation from direct access by any

operations (e.g. procedures or functions) not defined as part of the type definition. This feature is valuable in protecting the representation from unauthorized or undesirable modifications by other parts of the program. It is also important in aiding the verification that data objects maintain their specified properties even when the data representation is modified (authorized modification). The use of abstract data types to facilitate program modifications is treated more fully by Linden [Linden 76].

More recently, various programming language constructs have been suggested to support modularization, to deal with data type abstraction in a protected and controlled manner, and to support the generation of well-structured programs. Liskov and Zilles [Liskov 74] have developed a language CLU that implements function clusters. A cluster can guarantee integrity of its data objects because it controls their creation and access to an object is only through operations defined by the cluster. Wulf has introduced the concept of form, a mechanism for abstraction, into Alphard [Wulf 74, 76]. Forms are a somewhat more general structure than clusters: their major use are in describing an abstract data structure, the operations upon it and the permitted accesses to its components. The language Gypsy, developed at the University of Texas, is a language for specifying and implementing programs which are rigorously verifiable [Ambler 76a, 76b]. Gypsy uses a type unit that includes an access list associated with the object to realize the concept of abstract data types and explicitly state the access to their internal representation. The language Euclid was designed to facilitate the development of verifiable systems programs [Lampson 76]. Hierarchical nesting of program units is a feature of Euclid; type definitions are used with import and export lists (and suitable restrictions) to permit the definition of abstract data objects and to restrict access to their internal representations. At the time of this writing, the Euclid report issued was in draft form and not to be circulated. Since some of the issues related to abstract data types may require modification in the final language definition, the report will not be quoted. It is included in the survey to indicate another language design that has been influenced by the goals of verification and security.

CLU was recently designed at MIT as a language "to support structured programming". Procedures, which support functional abstractions, and operation clusters, which support abstract data types, are provided in the language. The compilation of the cluster defines the meaning of the type and the operations on that type (functions in the cluster) completely characterize the data types. The representation of the cluster is not accessible outside the cluster. A cluster definition looks like this:

<cluster-name>: cluster [(<cluster-Parameters>)] is <operator-list>

where

<cluster-name>	is the name of the abstract type being defined by the cluster
<cluster-parameters>	is the information that needs to be made available when objects of the abstract type are created
is	indicates that the data type defined corresponds to a set of operations
<operator-list>	is the set of operations that may be performed on objects of that type.

The other information provided by the cluster definition is:

(i) a description of the representation of objects of the abstract type

(ii) code which is to be executed when the objects are created

(iii) set of operator definitions.

The cluster functions are the means for accessing the data type representation. [Liskov 74].

Alphard, recently developed at Carnegie-Mellon University, was designed to support the development of well-structured programs and to facilitate their verification. Forms are the abstraction mechanism provided in the language. A form definition looks like:

```
form <form-name> (form-parameters) =

    beginform
      specifications . . . ;
      representation . . . ;
      implementation . . . ;
    endform;
```

where

<form-name>	is the name of the abstract type being defined by the form
<form-parameters>	is the known or desired information to be made available when objects of the abstract type are created
specifications . . .	is the information about the form necessary for its users -- it defines a set of accesses to objects of the type being defined by the form
representation . . .	defines the concrete representation and related properties of an object of the abstract type

implementation . . . is the definitions of the functions that may be applied to an object of the abstract type

A key feature of Alphard facilitating the verification of properties of a form is the separation of the use of the abstract type defined by the form from the definition of its representation. This feature permits the verification of the form independent of the abstract program which uses the form. This keeps the verification task manageable and minimizes verification resulting from program changes [Wulf 76]. Similar benefits can be cited for the other languages described.

Languages such as Simula 67, CLU and Alphard have been suggested as suitable environments for embedding an access control facility [Jones 76]. In order to incorporate access control in the language (which can be checked at compile-time) a type definition is extended to specify a set of rights. The user of a type's operation must have the appropriate rights to the objects passed as parameters to the operation. The definition of the proposed access control facility includes a binding rule determining how variables are bound to objects.

Gypsy, recently developed at the University of Texas, was designed as the core of an entire methodology for the development of programs which can be designed, specified, implemented and verified. Key features are the integration of programming and formal specification facilities into the language and complementary approaches to verification: formal proofs that the program meets its specifications, validation of specifications by run-time evaluation and a post-execution program trace. Pascal, suitably restricted and modified, served as the basis for the development of Gypsy. In particular, the hierarchical nesting of Pascal and non-local variables were eliminated; access lists were added. Gypsy deals with data abstraction, modularity and access control in the following manner. A program consists of units such as "routine" and "type". Access to the internal representation of a type, for example, is explicitly stated in an access list associated with the object. The absence of non-local variables simplifies the verification task and aids the incremental development of programs. Similar to the other languages discussed, the separation of the use of an abstraction from its implementation facilities the verification process.

CONCLUSIONS

The intent of this survey is to briefly show how the goal of achieving correct and secure software has impacted the design of programming languages for developing the software. Languages designed to explicitly support the construction of well-structured programs and to ease the task of verifying that they meet their formal specifications have only recently come

into existence. Serious exploitation of these languages in representative problem domains is essential before any significant conclusions regarding their utility can be drawn. Many open questions still remain. The security applications described in the paper are suggested as one possible domain of investigation.

Obviously, the principles of abstraction, modularity and access control can be applied to almost any programming language. However, the idea is to provide a language which explicitly supports them and therefore encourages their use. (A determined programmer can circumvent and misuse these facilities, of course.) After all, programming is a problem domain as valid as any other and it needs appropriate support from a programming language.

REFERENCES

Ambler 76a — A. L. Ambler, D. I. Good and W. F. Burger
Report on the Language CYPSY
University of Texas at Austin, Department of Computer Sciences, June 1976

Ambler 76b — A. L. Ambler, D. I. Good, J. C. Browne, W. F. Burger, R. M. Cohen, C. G. Hoch and R. E. Wells
CYPSY: A Language for Specification and Implementation of Verifiable Programs
University of Texas at Austin, Department of Computer Sciences, August 1976

Dahl 68 — O. J. Dahl, B. Myhrhaug and K. Nygaard
The Simula 67 Common Base Language
Norwegian Computer Center, Oslo, Norway, 1968

Dennis 75 — J. B. Dennis
An Example of Programming with Abstract Data Types
SIGPLAN Notices Vol. 10 No. 7 July 1975

Fisher 75 — P. Fisher, W. Hankley and J. McCall (editors)
Steps Toward Reliable Software: Proceedings of A Workshop
U. S. Army Computer Systems Command, Fort Belvoir, VA, Technical Documentary Report, January 1975 (Workshop held 19-20 November 1974 in Falls Church, VA; report prepared by Department of Computer Science, Kansas State University, Manhattan, Kansas)

Flon 75 — L. Flon
Program Design with Abstract Data Types
Carnegie-Mellon University, Department of Computer Science, June 1975

Hoare 72 — C. A. R. Hoare
Proof of Correctness of Data Representations
Acta Informatica, 1, 4 (1972) pp. 271-281

Jones 76 — A. K. Jones and B. H. Liskov
An Access Control Facility for Programming Languages
Carnegie-Mellon University, Department of Computer Science, May 1976

Lampson 76 — B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchell and G. J. Popek
Euclid Report (Draft)
April 1976

Linden 76 — T. A. Linden
The Use of Abstract Data Types to Simplify Program Modifications
Proc. of Conference on Data: Abstraction, Definition and Structure, March 1976

Liskov 74 — B. H. Liskov and S. Zilles
An Approach to Abstraction
Proc. of Symposium on Very High Level Languages, SIGPLAN Notices Vol. 9 No. 4 April 1974

Liskov 75 — B. H. and S. Zilles
Specification Techniques for Data Abstraction
Proc. of 1975 International Conference on Reliable Software, SIGPLAN Notices Vol. 10 No. 6 June 1975

Morris 73 — J. H. Morris
Protection in Programming Languages
CACM Vol 16, No. 1 January 1973

Palme 73 — J. Palme
Protected Program Modules in Simula 67
Research Institute of National Defense, Stockholm, Sweden, July 1973

Parnas 72a — D. L. Parnas
A Technique for Software Module Specification with Examples
CACM Vol. 15, No. 5 May 1972

Parnas 72b — D. L. Parnas
On the Criteria to be Used in Decomposing Systems into Modules
CACM Vol 15. No. 12 December 1972

Spitzen 76 — J. M. Spitzen, K. N. Levitt and L. Robinson
An Example of Hierarchical Design and Proof
SRI Report, Project 4097, March 1976

Wegbreit 74 — B. Wegbreit
The Treatment of Data Types in ELL
CACM Vol 17. No. 5 May 1974

Wirth 71 — N. Wirth
Program Development by Stepwise Refinement
CACM Vol 14. No. 4 April 1971

Wulf 74 — W. A. Wulf
ALPHARD: Toward a Language to Support Structured Programs
Carnegie-Mellon University, Department of Computer Science, April 1974

Wulf 76 — W. A. Wulf, R. L. London and M. Shaw
Abstraction and Verification in ALPHARD: Introduction to Language and Methodology
Carnegie-Mellon University, Department of Computer Science, June 1976

SYMBOLIC EXECUTION: A COMPROMISE BETWEEN TESTING AND VERIFICATION

JOHN A. DARRINGER

IBM T.J. WATSON RESEARCH CENTER
YORKTOWN HEIGHTS, NEW YORK

PROBLEM: HOW CAN WE BE CONFIDENT A PROGRAM WILL DO WHAT WE WANT?

IDEALLY: "PROVE IT CORRECT"

REQUIRES:

- SPECIFICATION OF DESIGNER'S INTENT
- LOOP INVARIANTS
- MODEL OF LANGUAGE SEMANTICS
- PROBLEM DOMAIN KNOWLEDGE

PROVIDES:

- MAXIMUM CONFIDENCE THAT SPECIFICATION IS CORRECTLY IMPLEMENTED

BUT IS THE SPECIFICATION CORRECT?

PRACTICE: TEST IT

"... PRESENCE NOT ABSENCE ..." BUT

REQUIRES LITTLE:

- COMPILER EMBODIES PL SEMANTICS (EXACT)
- SPECIFICATION
- INVARIANTS
- ADDITIONAL KNOWLEDGE

} ALL IN THE MIND OF THE TESTER

PROVIDES:

- SOME CONFIDENCE THAT SPECIFICATION IS CORRECTLY IMPLEMENTED
- SOME CONFIDENCE THAT SPECIFICATION IS SOUND

ALTERNATIVE: TEST IT SYMBOLICALLY

- CALL PROGRAM WITH SYMBOLIC VALUES

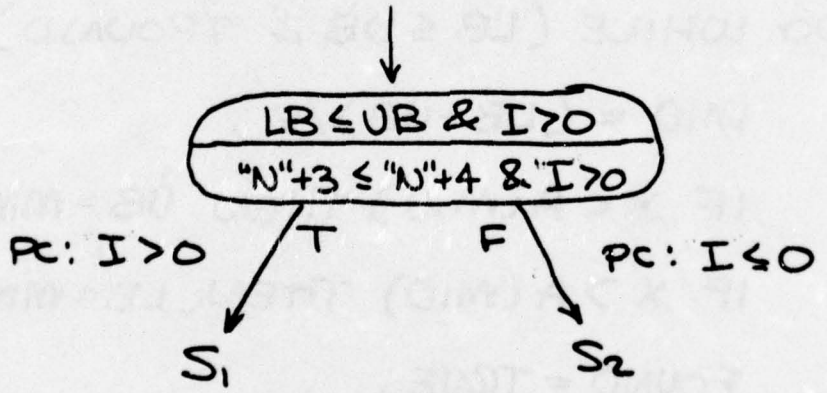
  CALL SEARCH ("A"A , "N" , "N"+4 , "X");

- VARIABLES HAVE FORMULAE AS VALUES

	LB IS "N" , UB IS "N"+4
MID = (LB+UB)/2 ;	MID IS "N"+2
LB = MID+1 ;	LB IS "N"+3

- CONDITIONAL STATEMENTS MAY PRODUCE BRANCHES IN SYMBOLIC EXECUTION TREE

  IF LB≤UB & I>0 THEN S_1; ELSE S_2;



- ONE SYMBOLIC TEST TYPICALLY CORRESPONDS TO AN ARBITRARLY NUMBER OF CONVENTIONAL TESTS

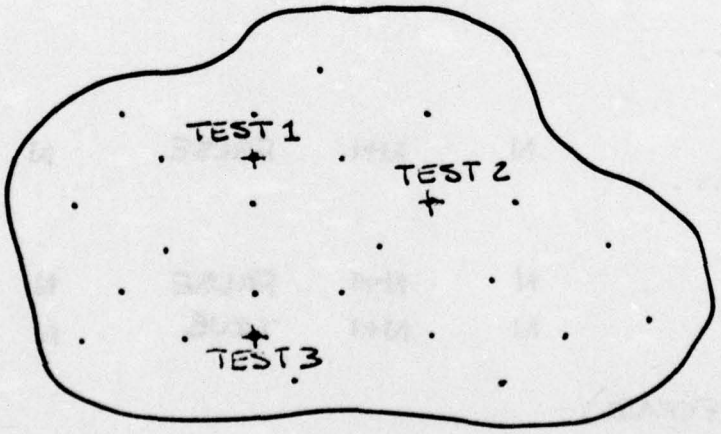
EXAMPLE: BINARY SEARCH ?




```
SEARCH: PROCEDURE (A,LB,UB,X);
        DCL  A(*) INTEGER,
             (LB,UB,X,MID) INTEGER,
             FOUND  BIT;
        FOUND = FALSE;
        DO WHILE (LB≤UB & ¬FOUND);
/* MID POINT */     MID = (LB+UB)/2;
/* LOOK LEFT */     IF X<A(MID) THEN UB=MID-1; ELSE
/* LOOK RIGHT */    IF X>A(MID) THEN LB=MID+1; ELSE
/* FOUND IT */      FOUND=TRUE;
                    END;
        DISPLAY (FOUND,MID);
        END;
```

TESTING BINARY SEARCH

- SELECT PARTICULAR VALUES
  - ARRAY LENGTH LB, UB
  - ARRAY VALUES A(LB), ··· A(UB)
  - SEARCH VALUE X
- CALL SEARCH
- CHECK RESULTS
- REPEAT WHILE >0
- EACH TEST CHECKS ONE POINT IN INPUT SPACE THAT IS ARBITRARILY LARGE IN SEVERAL DIMENSIONS



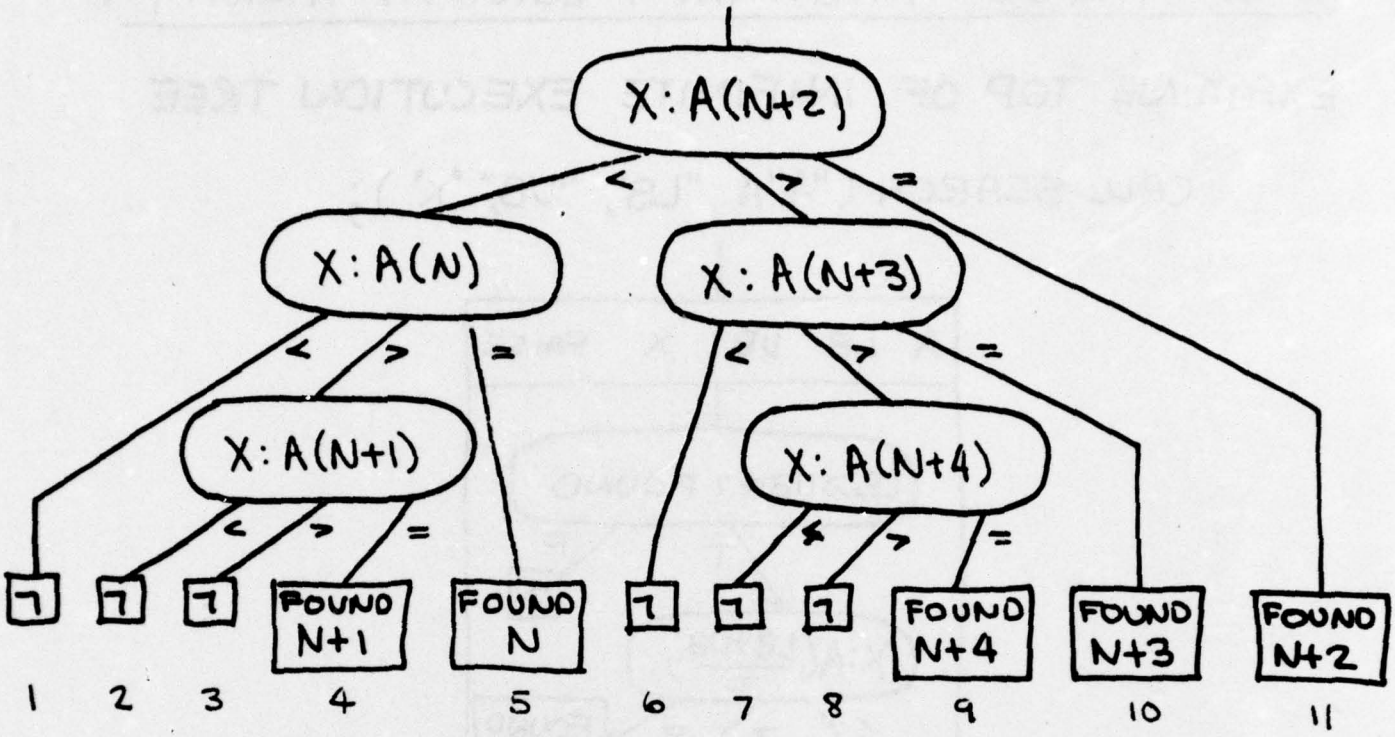
INPUT SPACE

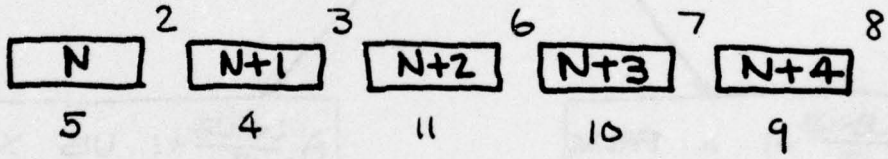
SYMBOLICALLY TESTING BINARY SEARCH

FIXED LENGTH ARRAY

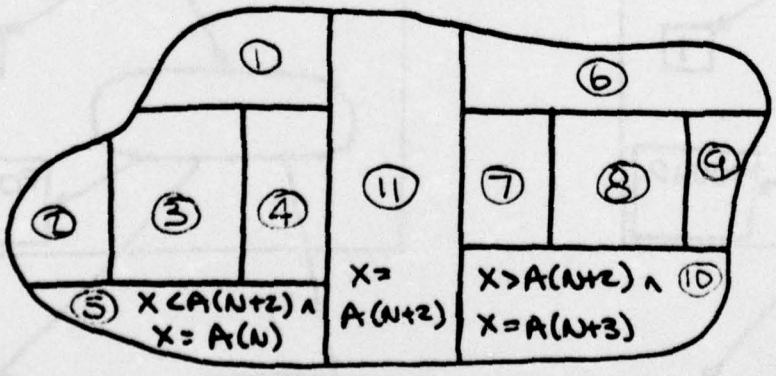
CALL SEARCH("A"A, "N", "N"+4, "X");	LB	UB	FOUND	MID	PC
	N	N+4	·	·	TRUE
FOUND = FALSE;	N	N+4	FALSE	·	TRUE
DO WHILE (LB≤UB & ¬FOUND)… TRUE					
MID = (LB+UB)/2;	N	N+4	FALSE	N+2	TRUE
IF X<A(MID) THEN …					
ASSUME TRUE	N	N+4	FALSE	N+2	X<A(N+2)
UB = MID-1;	N	N+1	FALSE	N+2	X<A(N+2)
DO WHILE (LB≤UB & ¬FOUND)… STILL TRUE					
MID = (LB+UB)/2;	N	N+1	FALSE	N	X<A(N+2)
IF X<A(MID) THEN …					
ASSUME FALSE	N	N+1	FALSE	N	X<A(N+2) & X≥A(N)
IF X>A(MID) THEN …					
ASSUME FALSE	N	N+1	FALSE	N	X<A(N+2) & X=A(N)
FOUND = TRUE;	N	N+1	TRUE	N	SAME
DO WHILE (LB≤UB & ¬FOUND)… FALSE					
DISPLAY (FOUND, MID);	N	N+1	TRUE	N	X<A(N+2) & X=A(N)

SYMBOLIC EXECUTION TREE (LENGTH = 5)





11 SYMBOLIC TESTS EXHAUST INPUT SPACE (LENGTH = 5)

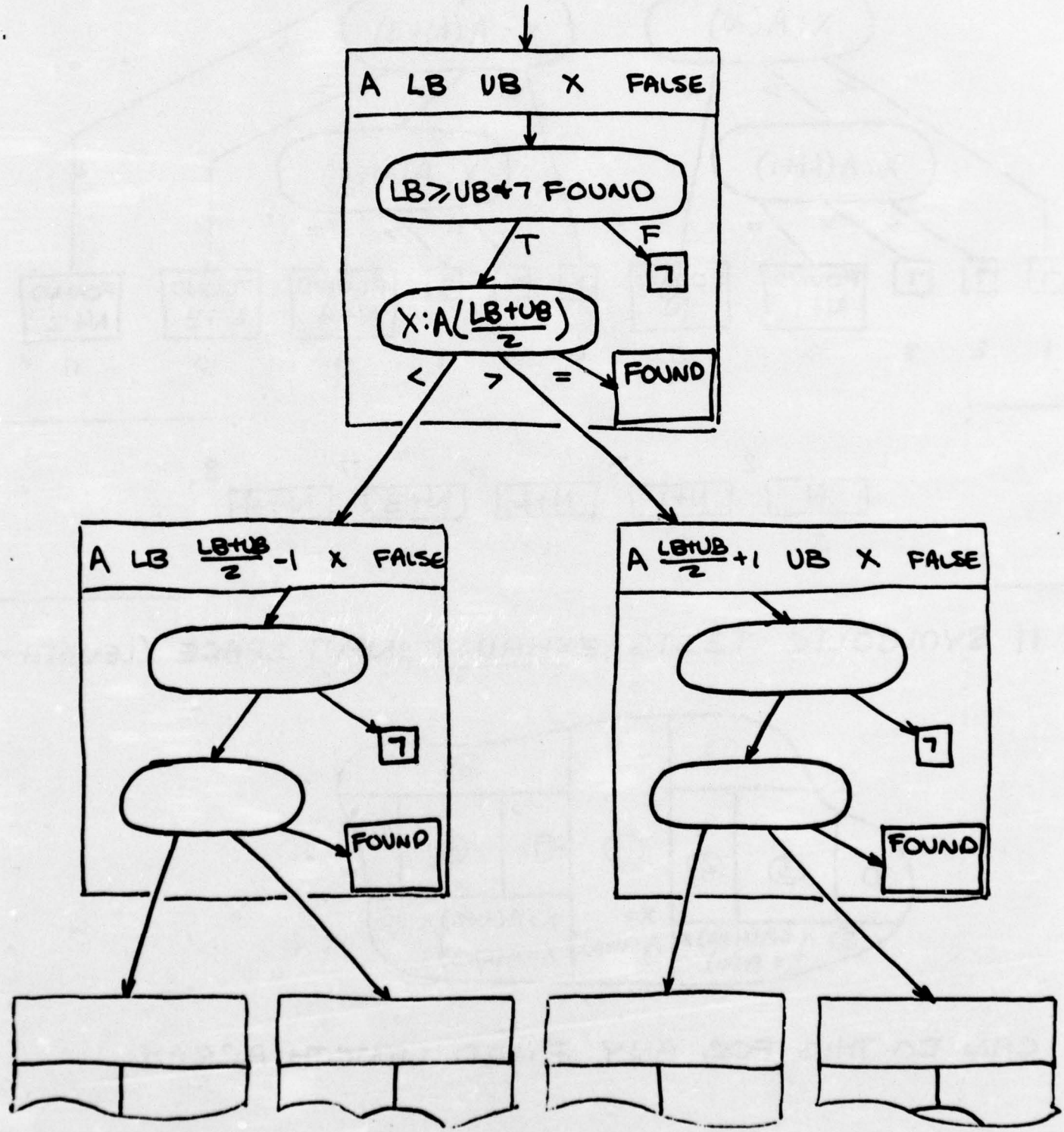


CAN DO THIS FOR ANY FIXED LENGTH ARRAY

WHAT ABOUT ARBITRARY LENGTH ARRAY?

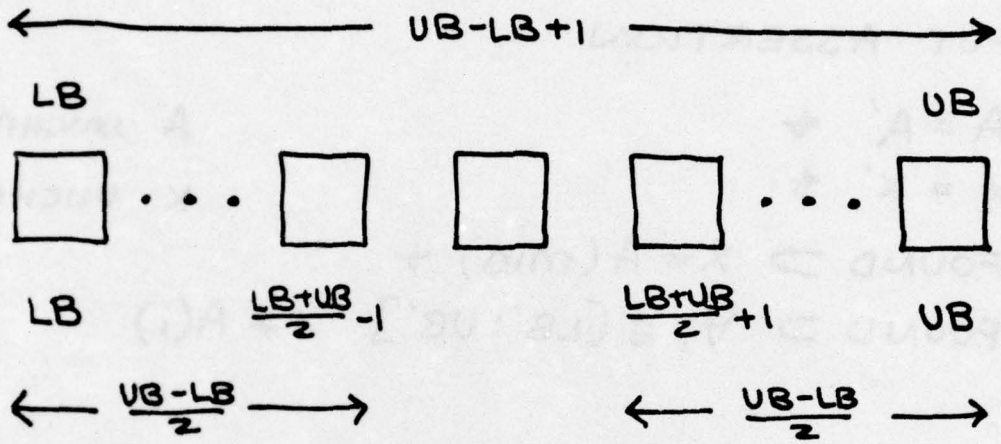
EXAMINE TOP OF INIFINITE EXECUTION TREE

CALL SEARCH ("A"A , "LB", "UB," "X");



INFORMAL INDUCTION

PERHAPS TESTER WILL DETECT PATTERNS IN THE EXECUTION TREE AND MAKE INDUCTIVE ARGUMENTS.



SUCH AS:

- SEARCH WORKS FOR LENGTH 5
- IF IT WORKS FOR LENGTH $\frac{UB-LB}{2}$
  THEN IT WILL WORK FOR LENGTH UB-LB+1
- THEREFORE ...

VERIFYING BINARY SEARCH:

- INPUT ASSERTION

  $\forall i \in [LB' : UB'-1] \quad A(i) \leq A(i+1)$ — A SORTED

- OUTPUT ASSERTION

  $A = A'$ ∧ — A UNCHANGED

  $X = X'$ ∧ — X UNCHANGED

  $FOUND \supset X = A(mid)$ ∧

  $\neg FOUND \supset \forall i \in [LB' : UB'] \quad X \neq A(i)$

- LOOP INVARIANT

  $A = A'$ ∧

  $X = X'$ ∧

  $\forall i \in [LB' : LB-1] \quad X \neq A(i)$ ∧ — X NOT LEFT OF LB

  $\forall i \in [UB+1 : UB'] \quad X \neq A(i)$ ∧ — X NOT RIGHT OF UB

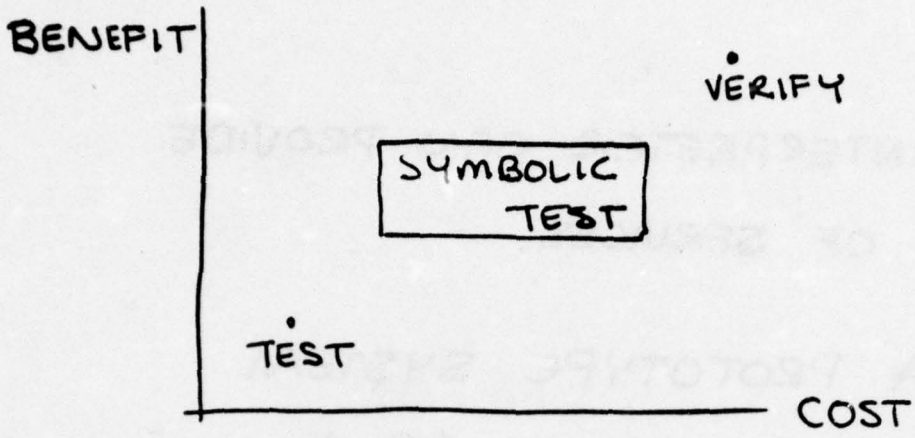
  $\forall i \in [LB' : UB'-1] \quad A(i) \leq A(i+1)$

- LEMMAS

  $\forall i \in [LB : UB-1] \quad A(i) \leq A(i+1)$ ∧

  $X < A(j)$ ∧ $j \in [LB : UB-1]$

  $\supset \forall i \in [j : UB-1] \quad X \neq A(i)$



SYMBOLIC TESTING

- PROVIDES MORE CONFIDENCE THAN TESTING, BUT REQUIRES A SYMBOLIC INTERPRETER.

- REQUIRES NO FORMAL SPECIFICATION OR INVARIANTS, BUT PROOFS ARE INFORMAL

SPECTRUM

A SYMBOLIC INTERPRETER CAN PROVIDE A SPECTRUM OF SERVICES.

EFFIGY - A PROTOTYPE SYSTEM

PROGRAMMING LANGUAGE (PL/I-like)

- INTEGERS, BITS {SCALAR, ARRAY} {STATIC, AUTO}
- + - * / ** MOD ABS & | ¬
- ASSIGNMENT
- IF... THEN... ELSE ...
- DO... END
- ITERATIVE DO, DO WHILE ... END
- GO TO AND LABELS
- PROCEDURES RECURSIVE
  PL/I PARAMETER CONVENTIONS

DEBUGGING

- TRACE
- BREAK POINTS
- STATE SAVING

EFFIGY

SYMBOLIC EXTENSIONS

- CONSTANTS "X", "ABC"A , "?X"
- PATH SELECTION

  GO TRUE , GO FALSE

  ASSUME (P), GO
- DISPLAY

SERVICES

• CALL P(1,3);	CONVENTIONAL EXECUTION
• CALL P("X","4");	SYMBOLIC EXECUTION
• CALL P("X", 3);	MIXED
• VERIFY VP;	VERIFICATION
• TEST (200) P;	DEPTH LIMITED GENERATION OF EXECUTION TREE
• COVER P(1,3);	FIND COVERAGE OF TEST CASE

TEST CASE COVERAGE

- GIVEN A SET OF TEST CASES $\{T_i\}$
  USE COVER TO FIND PREDICATES $\{P_i\}$
- IF $P_i = P_j$ THEN BOTH TESTS FOLLOW SAME CONTROL FLOW PATH, SAME SYMBOLIC TEST
- $P = \bigvee_{i=1}^{N} P_i$ INDICATES COVERAGE OF TESTS

TEST CASE GENERATION

- $\neg P$ INDICATES PART OF INPUT SPACE NOT COVERED
- FIND ANY SOLUTION TO $\neg P$ FOR NEW TEST

SUMMARY

- SYMBOLIC TESTING IS AN ALTERNATIVE TO TESTING AND VERIFICATION
- A SYMBOLIC INTERPRETER CAN PROVIDE A SPECTRUM OF SERVICES INCLUDING TESTING AND VERIFICATION

CURRENT CONCERNS:

LARGER CLASS OF DATA TYPES

- CHARACTERS
- POINTERS, BASED STRUCTURES

  NOT HARD BUT MAY NOT TELL MUCH?

SYMBOLIC EXECUTION OF SPECIFICATIONS

- MORE ECONOMIC VERIFICATION, EXECUTION
- EARLIER VERIFICATION, EXECUTION

PROCEDURAL HIERARCHY

EFFIGY CAN USE I/O SPEC IN VERIFICATION, EXECUTION

ABSTRACTION HIERARCHY

LOOKING AT

- TECHNIQUES FOR SPECIFYING DATA ABSTRACTIO
- EXECUTION OF THESE SPECIFICATIONS
- PROVING LOWER LEVEL CODE AND SPECS CORRECTLY IMPLEMENT HIGHER LEVEL

SELF-METRIC AND SELF-CHECKING SOFTWARE AND VALIDATION

S. S. Yau and R. C. Cheung

Department of Computer Sciences
Northwestern University
Evanston, Illinois 60201

Summary

The knowledge of the dynamic behavior of a program is extremely useful to its validation, especially for a large-scale program. Many of the current limitations on software validation and performance evaluation (performance testing may be considered a part of validation) are due to the lack of understanding of the dynamic behavior of the software because the behavior of a large-scale program is data-dependent and environment-dependent. It is desirable for a program to be able to automatically measure its own dynamic behavior. A program with such capabilities is called a piece of self-metric software.[1,2,3] Self-metric software is useful for evaluation of effectiveness of a test, the testedness of various parts of a program, detection of certain anomalies, generation of output validation tests and performance evaluation and optimization.

Self-metric software was first investigated and implemented in the SNUPER Computer by Estrin, et al.[4] in the mid-60's. In this system, they used hardware monitoring and software sampling techniques as well as self-metric software approach to measure equipment utilization, instruction-type or resident-routine-type usage, program execution activity at the source statement level and machine language level. Recently, self-metric software has been used extensively in automated evaluation and validation systems, such as PACE[5], PET[1,2] and JAVS[6,7].

There are many advantages for using self-metric software. The implementation of software probes requires little support or modification of the host computer system, and because software probes are normally implemented in the source language, the implementation can be quite convenient and flexible. There are primarily two types of self-metric techniques: measuring the frequency of execution and data values. One of the most important measurements is the program frequency path frequency which can be used for both program optimization and program testing. Some recent results in optimal placement of counters for measuring program path frequencies[8] will be discussed.

Self-metric software is used for collecting statistics of dynamic program behavior and hence plays a passive role in program validation. An extension of self-metric software is to incorporate the capability of checking the program dynamic behavior automatically during its execution. A piece of software with such capability is called self-checking software.[9] Although self-checking software has been primarily used to verify the correct operation of a computer system during execution time, especially for those real-time systems where ultra-reliability is required, it plays an active role in program validation during software development by isolating errors, test generation and validation.

There are also many advantages for using self-checking software.[9] For instance, it can be implemented at various program language levels ranging from design language, high-level language, machine language to microprogram. It checks hardware faults in addition to software errors, and can serve as security safeguards against malicious intrusion. It can be applied to various

levels of system design, such as system level, module level, instruction level and data level. It can also be applied to various stages of software development ranging from design, coding to testing. There are three major types of self-checking techniques, functional checking, control sequence checking[10] and data checking.

Although self-metric and self-checking techniques have been used extensively in certain software systems, there is a great need for developing formal methodology for design and generating cost-effective self-metric software and self-checking software. In order to apply self-metric and self-checking software to program validation, the most important research needed is to identify what information on the program behavior is most useful for its validation and also easy to obtain.

References

1. L. G. Studki, "Automatic Generation of Self-Metric Software," Proc. 1973 IEEE Symp. on Computer Software Reliability, New York, 1973, pp. 94-100.

2. L. G. Stucki and G. L. Foshee, "New Assertion Concepts for Self-Metric Software Validation," Proc. of 1975 International Conf. on Reliable Software, April 1975, pp. 59-65.

3. R. C. Cheung, K. H. Kim, C. V. Ramamoorthy, and S. S. Reddi, "Automated Generation of Self-Metric Software," 7th Hawaii International Conference on Systems Sciences, January, 1974, pp. 149-151.

4. G. Estrin, D. Hopkins, B. Coggan and S. D. Crocker, "SNUPER COMPUTER-a computer in instrumentation automation," Proc. 1967 Spring Joint Computer Conf., pp. 645-656.

5. J. R. Brown and R. H. Hoffman, "Evaluating the Effectiveness of Software Verification-Practical Experience with an Automated Tool," Proc. of the 1972 Fall Joint Computer Conf., pp. 181-190.

6. N. B. Brooks, C. Gannon, and R. J. Urban, JAVS Computer Program Documentation: User's Guide, General Research Corporation, Sept. 1975.

7. E. F. Miller, "Methodology for Comprehensive Software Testing," General Research Corporation CR-1-465, Rome Air Development Center RADC-TR-75-101, June 1975.

8. R. C. Cheung and C. V. Ramamoorthy, "Optical Measurement of Program Path Frequencies and its Applications," 1975 International Federation of Automatic Control Congress, August, 1975.

9. S. S. Yau and R. C. Cheung, "Design of Self-checking Software," Proc. 1975 International Conference on Reliable Software, pp. 450-457, April, 1975.

10. J. R. Kane and S. S. Yau, "Concurrent Software Fault Detection," IEEE Trans. on Software Engineering, Vol. 1, No. 1, pp. 87-99, March, 1975.

TESTING

TO INCREASE OUR CONFIDENCE THAT THE PROGRAM WILL EXECUTE THE SPECIFIED FUNCTIONS CORRECTLY WITH DESIRED PERFORMANCE.

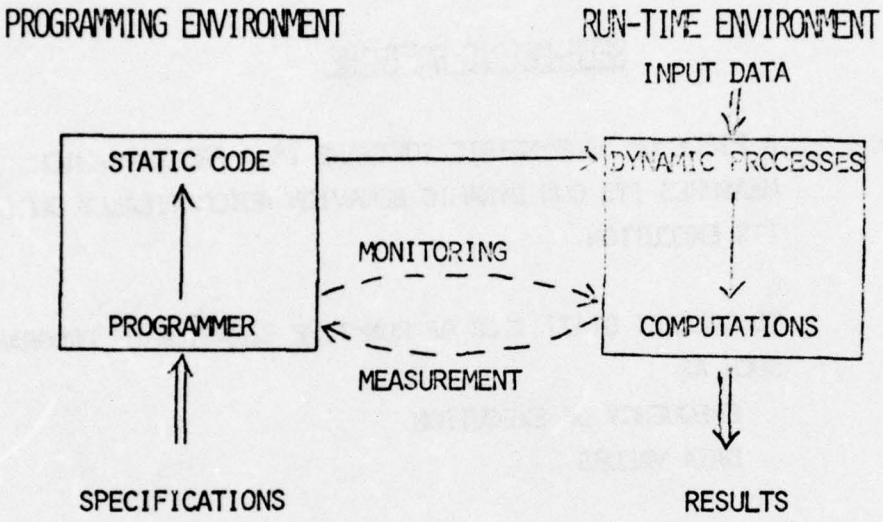
OBJECTIVES OF EFFECTIVE TESTING

1. TO DETECT AS MANY ERRORS (FUNCTIONAL OR PERFORMANCE DEFICIENCIES) AS POSSIBLE.
2. TO DETECT THE ERRORS AS EARLY AS POSSIBLE IN THE SOFTWARE DEVELOPMENT CYCLE.
   (I) SAVE SOFTWARE DEVELOPMENT COST
   (II) EASIER TO CORRECT AN ERROR
   BETTER PROGRAMMING PRODUCTIVITY AND PROGRAM RELIABILITY.
3. TO DEVELOP A COMPREHENSIVE SET OF ACCEPTANCE TESTS.

CHARACTERISTICS OF LARGE PROGRAMS

- COMPLEX STRUCTURE, AND MANY INSTRUCTIONS WITH COMPLICATED DATA STRUCTURES.
- DEVELOPED BY A LARGE NUMBER OF PROGRAMMERS, SOMETIMES IN DIFFERENT LOCATIONS.
- A SIGNIFICANT NUMBER OF FLOW PATHS.
- INDIVIDUAL MODULES ARE USUALLY THOROUGHLY TESTED, BUT NOT THE WHOLE SYSTEM. -- PROGRAM ERRORS CAN USUALLY BE TRACED BACK TO INCORRECT INTERACTIONS BETWEEN MODULES.
- PROGRAM BEHAVIOR IS DATA-DEPENDENT AND ENVIRONMENT-DEPENDENT.
- PROGRAM BEHAVIOR CANNOT BE RELIABLY PREDICTED.
- A LARGE AMOUNT OF 'DORMANT' SOFTWARE BUGS MAY STILL BE PRESENT AFTER THE PROGRAM IS PUT INTO OPERATION A LONG TIME
- FREQUENT MODIFICATIONS:
  - CORRECT AN ERROR JUST DISCOVERED,
  - SATISFY NEW SPECIFICATIONS AND REQUIREMENTS,
  - IMPROVE THE EFFICIENCY OF THE PROGRAM.
- EACH MODIFICATION MAY INTRODUCE UNEXPECTED SOFTWARE ERRORS.

AUTOMATED MEASUREMENT & MONITORING OF PROGRAM DYNAMIC BEHAVIOR



COMPUTATIONS ⟶ PROGRAMMER: INFORMATION ON RUN-TIME BEHAVIOR. (SELF-METRIC)

PROGRAMMER ⟶ COMPUTATIONS: MONITORING OF THE COMPUTATIONS PERFORMED. (SELF-CHECKING)

SELF-METRIC SOFTWARE

DEFINITION: A PIECE OF SELF-METRIC SOFTWARE IS A PROGRAM WHICH MEASURES ITS OWN DYNAMIC BEHAVIOR AUTOMATICALLY DURING ITS EXECUTION.

PURPOSES: TO COLLECT STATISTICS OF RUN-TIME BEHAVIOR OF PROGRAMS SUCH AS

- FREQUENCY OF EXECUTION
- DATA VALUES

APPLICATION:

* PROGRAM OPTIMIZATION
* PROGRAM VALIDATION

IMPLEMENTATION: INSERTION OF SOFTWARE PROBES INTO SOURCE PROGRAM AFTER AUTOMATED STATIC ANALYSIS

ADVANTAGES OF SELF-METRIC SOFTWARE

* SOFTWARE PROBES ARE IMPLEMENTED IN SOURCE LANGUAGE
  - CONVENIENT
  - FLEXIBLE
* HIGH DEGREE OF AUTOMATION POSSIBLE
* OPTIMIZATION POSSIBLE TO MINIMIZE INTERFERENCE TO PROGRAM OPERATION
* EFFECT OF INTERFERENCE PREDICTABLE
* EASY INSERTION AND REMOVAL OF SOFTWARE PROBES
* REQUIRES LITTLE SUPPORT OR MODIFICATION OF THE HOST COMPUTER SYSTEM

TYPES OF SELF-METRIC TECHNIQUES

FREQUENCY OF EXECUTION

* PROGRAM PROFILE - FREQUENCY OF EXECUTION OF EVERY STATEMENT
* DATA ACTIVITY PROFILE - FREQUENCY OF REFERENCE
  FREQUENCY OF ASSIGNMENT
  FREQUENCY OF A PARTICULAR OPERATION ON A PIECE OF DATA
* PROGRAM PATH FREQUENCY - FREQUENCY OF ACTIVATION OF A CONTROL SEQUENCE

MEASUREMENT OF DATA VALUES

* PROGRAM VARIABLES - INITIAL VALUE
  FINAL VALUE
  MINIMUM VALUE
  MAXIMUM VALUE
  MEAN
  VARIANCE
* PARAMETER VALUES

APPLICATIONS OF PROGRAM PATH FREQUENCIES

PROGRAM OPTIMIZATION

*NODE FREQUENCY MEASUREMENT PINPOINTS THE SEGMENTS OF CODE THAT ARE EXECUTED MOST OFTEN.

*PATH FREQUENCY MEASUREMENT PINPOINTS THE PROCESSES (CORRESPONDING TO PATHS IN A PROGRAM) THAT ARE EXECUTED MOST OFTEN.
OPTIMIZATION EFFORTS SHOULD CONCENTRATE ON THESE PARTS.

*PATH FREQUENCY MEASUREMENT GIVES US THE BRANCHING CHARACTERISTICS FOR EFFICIENT LOOK-AHEAD SCHEDULING, ESPECIALLY FOR PIPELINED MACHINES LIKE THE CDC6600. IT ALSO HELPS REGISTER ALLOCATION AND SUBROUTINE OPTIMIZATION.

*DETECTION OF DEAD CODE.

*OPTIMIZATION OF THE PLACEMENT OF FILES IN A SEQUENTIAL MEMORY STORAGE SYSTEM.

*PROGRAM RESTRUCTURING FOR VIRTUAL MEMORY SYSTEM WITH PAGING.

PROGRAM TESTING

*EVALUATE THE TESTEDNESS OF A PIECE OF SOFTWARE.

*DEAD CODE DETECTION

*ELIMINATE REDUNDANT TESTING BY PINPOINTING PORTIONS OF THE PROGRAM WHERE ADDITIONAL TESTS ARE NEEDED.

*HELPS TO DESIGN EFFECTIVE TESTS BY PATH SENSITIZATION.

OPTIMAL MEASUREMENT OF PROGRAM PATH FREQUENCIES

DIRECTED GRAPH MODEL OF PROGRAM STRUCTURE

A PROGRAM IS REPRESENTED BY A DIRECTED GRAPH SUCH THAT

A NODE REPRESENTS COMPUTATIONAL TASK (SEGMENT OF CODE)

AN ARC REPRESENTS A POSSIBLE TRANSFER OF CONTROL FROM A SEGMENT OF CODE TO ANOTHER.

DEFINITIONS

*A PATH IS A SEQUENCE OF ARCS.

*THE LENGTH OF A PATH IS THE NUMBER OF ARCS ON THE PATH.

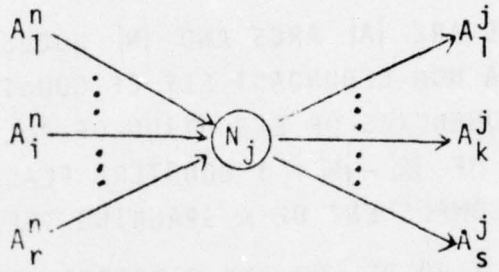
A NODE IS A PATH OF LENGTH ZERO.

AN ARC IS A PATH OF LENGTH ONE.

*A SPANNING TREE OF A DIRECTED GRAPH G IS A TREE (UNDIRECTED) WHICH CONTAINS ALL THE NODES OF G AND IS A SUBGRAPH OF G.

CONSERVATION OF FLOW AT A NODE

DEFINITION: $F(P)$ = FREQUENCY OF EXECUTION OF THE PATH P IN THE PROGRAM
= FLOW THROUGH THE PATH P



THE TWO FLOW CONSERVATION EQUATIONS FOR N_j ARE

$$\sum_{i=1}^{r} F(A_i^n) = F(N_j) = \sum_{k=1}^{s} F(A_k^j)$$

THEOREM 1

THE KNOWLEDGE OF THE FREQUENCIES OF EXECUTION OF ALL THE DIRECTED ARCS NOT IN A SPANNING TREE OF THE PROGRAM GRAPH ARE BOTH NECESSARY AND SUFFICIENT TO DETERMINE THE FREQUENCIES OF EXECUTION OF ALL THE DIRECTED ARCS IN THE PROGRAM GRAPH.

REMARKS:

- IF THERE ARE $|A|$ ARCS AND $|N|$ NODES IN A PROGRAM GRAPH, A NON-REDUNDANT SET OF COUNTERS TO MEASURE THE FREQUENCIES OF EXECUTION OF ALL THE ARCS WILL CONSIST OF $|A| - |N| + 1$ COUNTERS PLACED ON THE ARCS ON THE COMPLEMENT OF A SPANNING TREE.
- THEOREM 1 HOLDS FOR ANY DIRECTED GRAPH SUCH THAT CONSERVATION OF FLOW IS OBSERVED AT EVERY NODE.

ALGORITHM 1: OPTIMAL MEASUREMENT OF ARC FREQUENCIES

1. ASSIGN A COST C_i TO EVERY ARC IN THE GRAPH AS THE COST OF PLACING A COUNTER ON THE ARC A_i

$$C_i = K_i + d\ f_i$$

WHERE K_i = COST DUE TO STORAGE AND EXECUTION TIME INTERFERENCE BY MONITORING ARC A_i

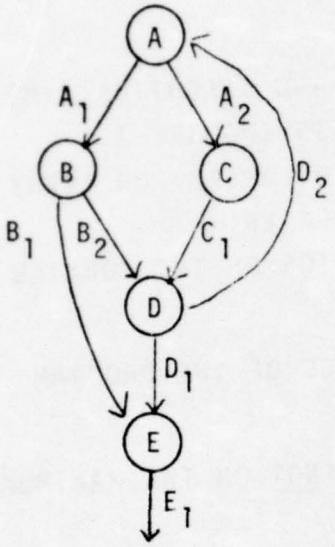
d = COST OF MEASUREMENT OVERHEAD FOR EVERY ACTIVATION OF THE COUNTER

f_i = FREQUENCY OF ACTIVATION OF THE COUNTER ON ARC A_i

2. FIND THE MAXIMUM (COST) SPANNING TREE OF THE PROGRAM GRAPH USING THE GREEDY ALGORITHM.

3. PLACE A COUNTER ON EACH OF THE ARCS NOT ON THE MAXIMUM SPANNING TREE.

AN EXAMPLE OF OPTIMAL ARC FREQUENCY MEASUREMENT



COUNTERS SHOULD BE PLACED ON THE ARCS B_1, B_2, C_1 AND E_1.

$F(A_1) = F(B_1) + F(B_2)$

$F(A_2) = F(C_1)$

$F(D_1) = F(E_1) - F(B_1)$

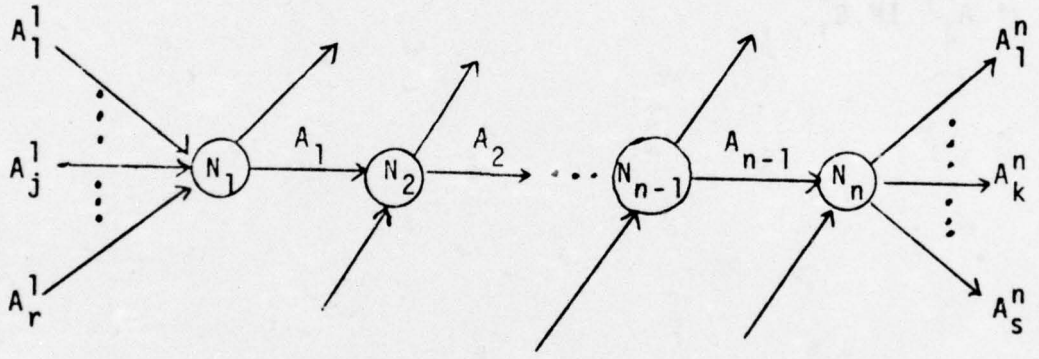
$F(D_2) = F(B_1) + F(C_1) - F(E_1) + F(B_2)$

THEOREM 2 (PATH FLOW CONSERVATION THEOREM)

FOR EVERY PATH P OF LENGTH n-1, $(N_1, N_2, \ldots, N_n)$, FROM NODE N_1 TO N_n IN A DIRECTED GRAPH G, WE CAN WRITE TWO FLOW CONSERVATION EQUATIONS FOR THE FLOW OF ALL THE PATHS OF LENGTH n THAT CONTAIN P AS A SUBPATH:

$$\sum_{j=1}^{r} F(A_j^1 P) = F(P) = \sum_{k=1}^{s} F(PA_k^n)$$

WHERE A_j^1 (FOR j=1, ..., r) ARE THE DIRECTED ARCS ENTERING N_1 AND A_k^n (FOR k=1, ..., s) ARE THE DIRECTED ARCS LEAVING N_n.



DEFINITION

THE i^{th} ORDER GRAPH OF A DIRECTED GRAPH G, G_i^*, IS A DIRECTED GRAPH THAT REPRESENTS THE RELATIONSHIP OF THE FLOW OF THE PATHS OF LENGTH i IN G WITH FLOW CONSERVATION OBSERVED AT EVERY NODE. THEREFORE AN ARC IN G_i^* REPRESENTS A PATH OF LENGTH i AND A NODE A PATH OF LENGTH i-1 IN G.

ALGORITHM 2: CONSTRUCTION OF G_{i+1}^* FROM G_i^*

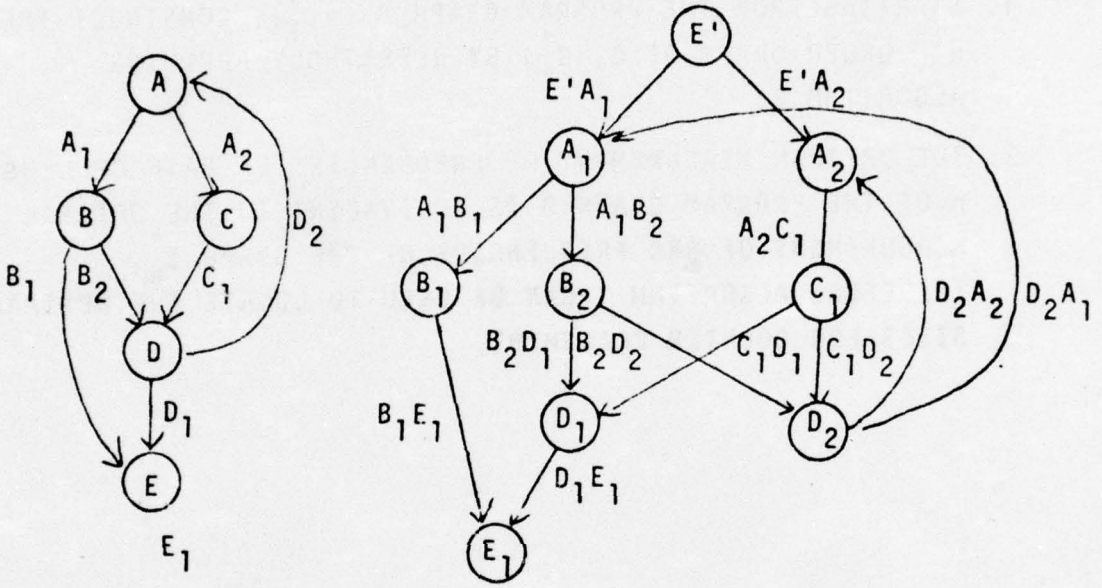
1. FOR EVERY ARC A_j^{*i} IN G_i^* CREATE A NODE N_j^{*i+1} TO REPRESENT IT IN G_{i+1}^*.
2. AN ARC (N_j^{*i+1}, N_k^{*i+1}) IS CREATED IN G_{i+1}^* IF A_k^{*i} FOLLOWS A_j^{*i} IN G_i^*.

ALGORITHM 3: OPTIMAL MEASUREMENT OF FREQUENCIES OF PATH OF LENGTH n OF THE PROGRAM GRAPH G

1. STARTING FROM THE PROGRAM GRAPH G ($=G_1^*$), CONSTRUCT THE n^{th} ORDER GRAPH OF G, G_n^*, BY REPEATEDLY APPLYING ALGORITHM 2.

2. THE OPTIMAL MEASUREMENT OF FREQUENCIES OF PATH OF LENGTH n OF THE PROGRAM GRAPH G IS EQUIVALENT TO THE OPTIMAL MEASUREMENT OF ARC FREQUENCIES OF THE GRAPH G_n^*. THEREFORE ALGORITHM 1 CAN BE USED TO LOCATE THE OPTIMAL SITES FOR COUNTER PLACEMENT.

AN EXAMPLE OF OPTIMAL PATH FREQUENCY MEASUREMENT (OF PATHS OF LENGTH 2)



(a) G (=G_1^*)

(b) G_2^*

COUNTERS SHOULD BE PLACED ON THE PATHS D_2A_2, C_1D_2, B_2D_1, B_2D_2, AND D_1E_1.

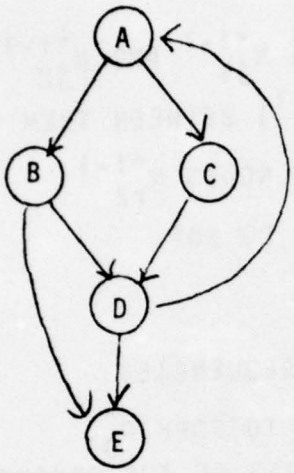
ALGORITHM 4: CONSTRUCTION OF G^*_{i-1} FROM G^*_i

1. FOR EVERY NODE N^{*i}_j IN G^*_i CREATE TWO NODES N^{*i-1}_{j1} AND N^{*i-1}_{j2} IN G^*_{i-1} WITH AN ARC $A^{*i-1}_j = (N^{*i-1}_{j1}, N^{*i-1}_{j2})$ BETWEEN THEM.
2. FOR EVERY ARC (N^{*i}_r, N^{*i}_k) IN G^*_i MERGE THE NODES N^{*i-1}_{r2} AND N^{*i-1}_{k1} TOGETHER INTO A NODE IN G^*_{i-1}. DO NOT ELIMINATE ANY ARC A^{*i-1}_j.

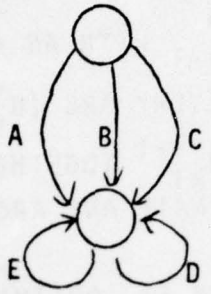
ALGORITHM 5: OPTIMAL MEASUREMENT OF NODE FREQUENCIES

1. APPLY ALGORITHM 4 ON THE PROGRAM GRAPH G TO FORM G^*_0.
2. THE OPTIMAL MEASUREMENT OF NODE FREQUENCIES OF THE PROGRAM GRAPH G IS EQUIVALENT TO THE OPTIMAL MEASUREMENT OF ARC FREQUENCIES OF THE GRAPH G^*_0.
   THEREFORE ALGORITHM 1 CAN BE APPLIED TO LOCATE THE OPTIMAL SITES FOR COUNTER PLACEMENT.

AN EXAMPLE OF OPTIMAL NODE FREQUENCY MEASUREMENT



(a) G



(b) G_0^*

COUNTERS SHOULD BE PLACED ON THE NODES A, B, D, AND E.

$F(C) = F(A) - F(B)$

SELF-CHECKING SOFTWARE

DEFINITION: A PIECE OF SELF-CHECKING SOFTWARE IS A PROGRAM WHICH CHECKS ITS OWN DYNAMIC BEHAVIOR AUTOMATICALLY DURING ITS EXECUTION.

PURPOSES:

* TO DETECT SOFTWARE ERRORS.
* TO ISOLATE SOFTWARE ERRORS.
* TO HELP THE SYSTEM TO RECOVER FROM ERRORS.
* TO VERIFY THE INTEGRITY OF THE SYSTEM.

APPLICATION:

* SELF-CHECKING SOFTWARE IS PRIMARILY DESIGNED TO VERIFY THE CORRECT OPERATION OF THE SYSTEM DURING EXECUTION TIME, ESPECIALLY REAL-TIME SYSTEMS WHERE ULTRA-RELIABILITY IS REQUIRED.
* IT IS ALSO USEFUL IN DETECTING AND ISOLATING ERRORS DURING SOFTWARE DEVELOPMENT.

IMPLEMENTATION:

1. BY INTRODUCING SOFTWARE REDUNDANCY (INSTRUCTION REDUNDANCY AND DATA REDUNDANCY) INTO THE SOFTWARE SYSTEM.
2. BY BUILDING SPECIALIZED HARDWARE TO PERFORM THE CHECKING.

ADVANTAGES OF SELF-CHECKING SOFTWARE

* IT CAN BE IMPLEMENTED AT DIFFERENT LEVELS:

  DESIGN LANGUAGE
  HIGH LEVEL LANGUAGE
  MACHINE LANGUAGE
  MICROPROGRAM

* CHECKING CAN BE PERFORMED CONCURRENTLY WITH PROGRAM COMPUTATION.

* IT CHECKS HARDWARE FAULTS IN ADDITION TO SOFTWARE ERRORS.

* IT CAN SERVE AS SECURITY SAFEGUARDS AGAINST MALICIOUS
  INTRUSION OR PROGRAMMER PLANTED BUGS.

* IT CAN IMPROVE THE AVAILABILITY OF THE SYSTEM.

SYSTEM DESIGN WITH SELF-CHECKING SOFTWARE

* SELF-CHECKING SOFTWARE CAN BE APPLIED TO DIFFERENT LEVELS
  - SYSTEM LEVEL
  - MODULE LEVEL
  - INSTRUCTION LEVEL
  - DATA LEVEL

* SELF-CHECKING SOFTWARE CAN BE APPLIED IN DIFFERENT STAGES OF SOFTWARE DEVELOPMENT
  - DESIGN
  - CODING
  - TESTING

* DEVELOPMENT OF SPECIALIZED HARDWARE TO IMPLEMENT EFFECTIVE CHECKING TECHNIQUES
  - HARDWARE TIMER
  - ASSOCIATIVE PROCESSOR
  - HARDWARE WATCHDOG

TYPES OF SELF-CHECKING TECHNIQUES

* FUNCTIONAL CHECKING
  - VERIFY REASONABLENESS OF OUTPUT
    - CHECK IMPOSSIBLE CONDITIONS
    - CHECK RELATIONSHIPS OF OUTPUT VALUES
  - VERIFY CONSISTENCY OF OUTPUT
    - SUBSTITUTION OF SOLUTIONS INTO EQUATIONS
    - REVERSE TRANSLATION
    - VERIFY CONSISTENCY OF RESOURCE STATE

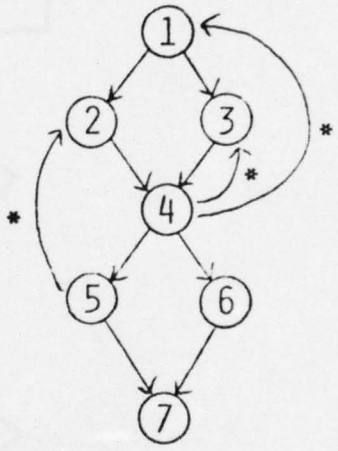
* CONTROL SEQUENCE CHECKING
  - INFINITE LOOP
    - MAXIMUM BOUND CHECKER
  - ILLEGAL BRANCH
    - RELAY-RUNNER SCHEME
    - POSITIVE CHECK
  - WRONG BRANCH
    - DECISION LOGIC IN SUCCESSOR MODULES
  - INCORRECT LOOP TERMINATION
    - SIMILAR TO WRONG BRANCH

* DATA CHECKING
  - INTEGRITY OF DATA VALUES
    - CHECKSUM
    - COMPARISON OF STATE OF RESOURCE TO SYSTEM TABLE
    - MEMORY PROTECTION MECHANISMS
    - COMPARISON WITH BACKUP COPY
  - INTEGRITY OF DATA STRUCTURES
    - LINKAGE CHECKS FOR LINKED LISTS
    - INFORMATION ON STATE OF DATA
  - NATURE OF DATA VALUES
    - DATA FILTER (MAXIMUM AND MINIMUM BOUNDS)
    - ASSERTIONS (RELATIONSHIPS BETWEEN DATA)

CONTROL SEQUENCE CHECKING --

TO DETECT ALL INFINITE LOOPS

1. GENERATE THE PROGRAM GRAPH.
2. DETERMINE ALL STRONGLY CONNECTED REGIONS (CYCLES)
3. DETERMINE THE MINIMUM NUMBER OF BRANCHES THAT NEE
   IN ORDER TO ELIMINATE ALL STRONGLY CONNECTED REGIC
4. INSERT A MAXIMUM BOUND CHECKER ON EACH OF THESE BR

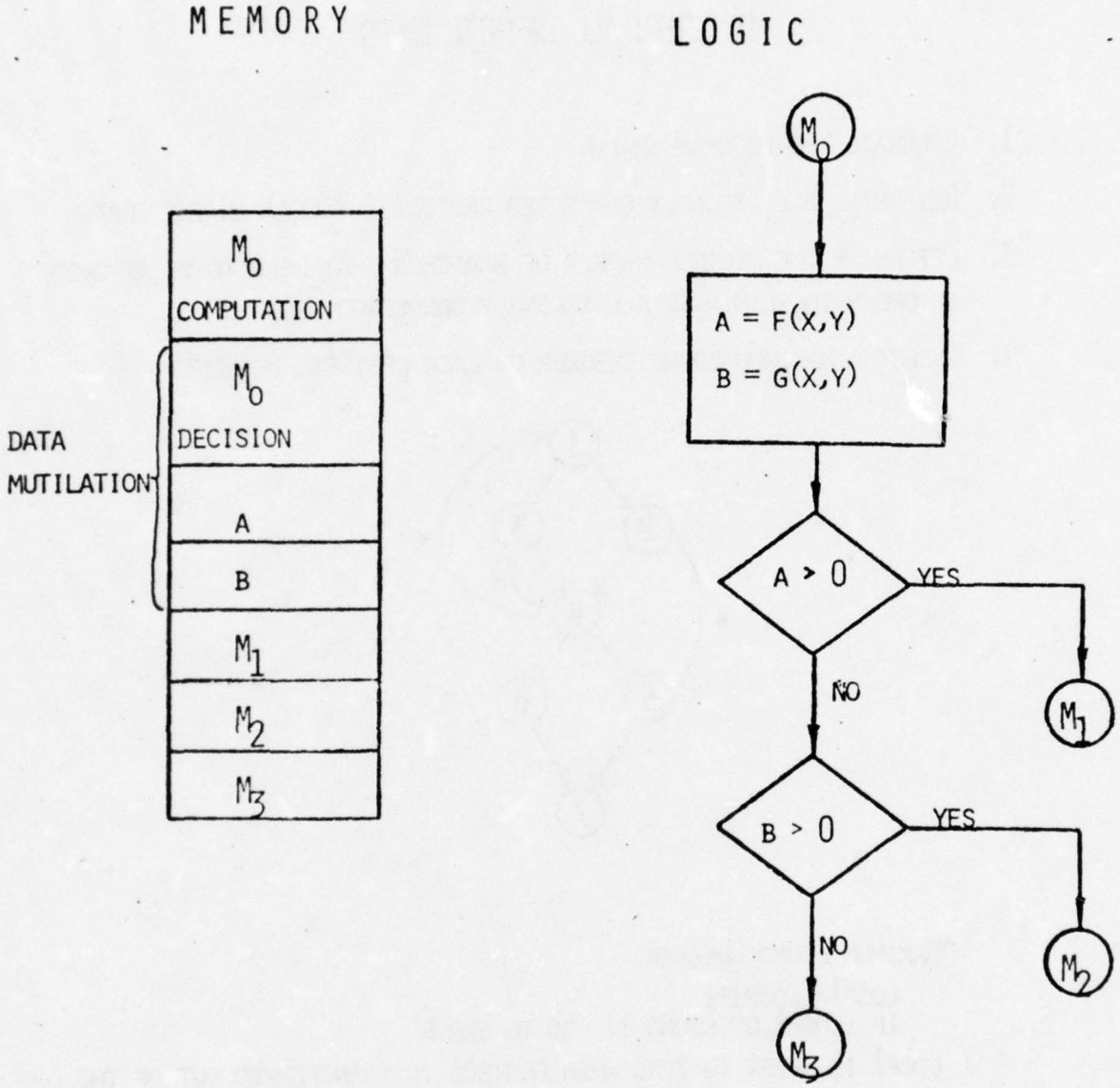


*MAXIMUM BOUND CHECKER

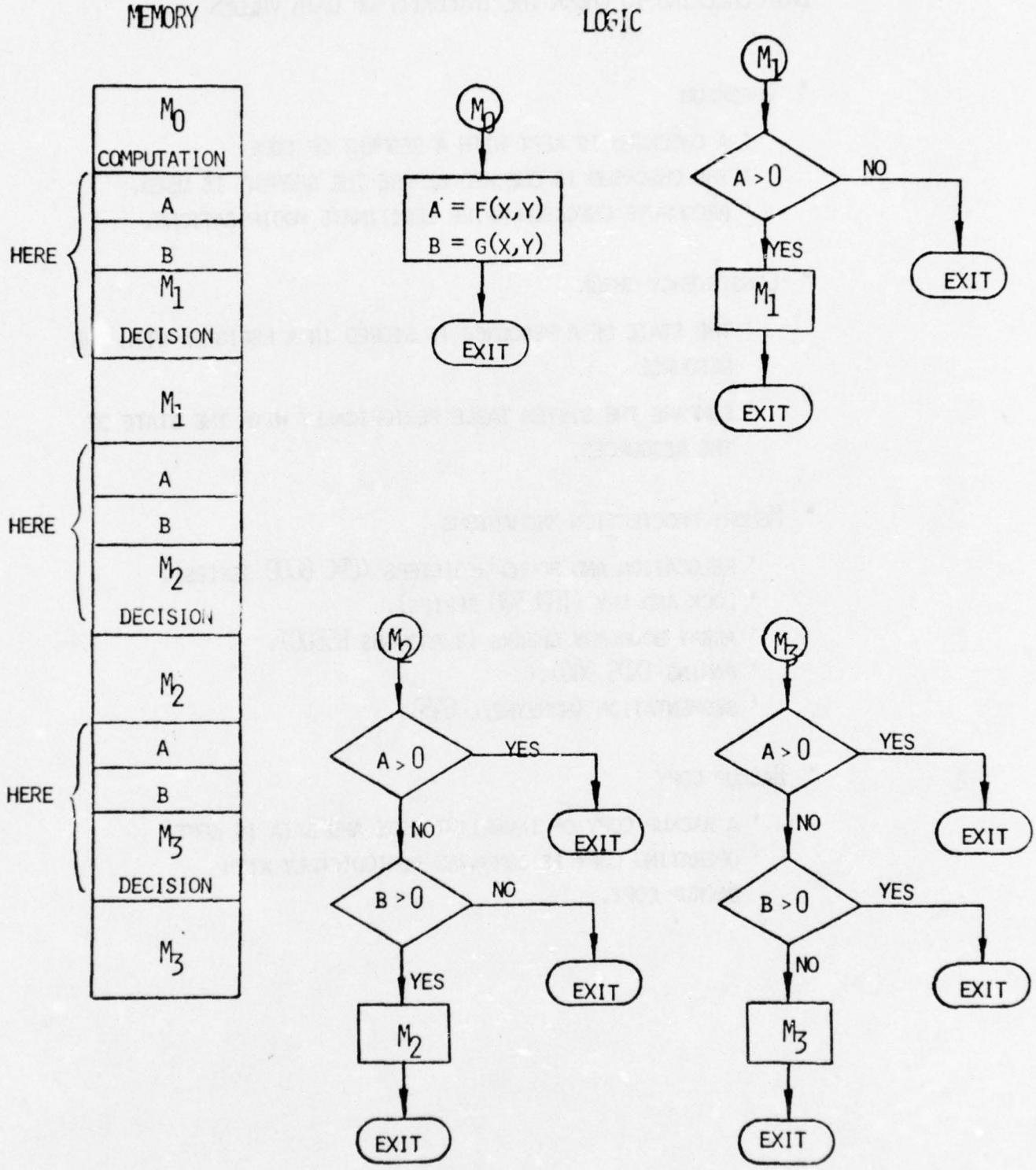
```
LOOP1←LOOP1+1
IF (LOOP1.GT.BOUND 1)  GO TO ERROR
```

LOOP1 IS RESET TO ZERO WHEN CONTROL IS TRANSFERR

(CONTROL SEQUENCE CHECKING -- TO DETECT WRONG BRANCHES)



CONVENTIONAL SCHEME



CONTROL SEQUENCE CHECKING - TO DETECT WRONG BRANCHES

DATA CHECKING-TO CHECK THE INTEGRITY OF DATA VALUES

* CHECKSUM
  - A CHECKSUM IS KEPT WITH A SEGMENT OF CODE.
  - THE CHECKSUM IS CHECKED BEFORE THE SEGMENT IS USED.
  - RECOMPUTE CHECKSUM AFTER LEGITIMATE MODIFICATIONS.

* CONSISTENCY CHECK
  - THE STATE OF A RESOURCE IS STORED IN A REGISTER WITH THE RESOURCE.
  - COMPARE THE SYSTEM TABLE PERIODICALLY WITH THE STATE OF THE RESOURCES.

* MEMORY PROCTECTION MECHANISMS
  - RELOCATION AND BOUND REGISTERS (CDC 6000 SERIES).
  - LOCK AND KEY (IBM 360 SERIES).
  - ARRAY BOUNDARY CHECKS (BURROUGHS B5500).
  - PAGING (XDS 940).
  - SEGMENTATION (HONEYWELL 645).

* BACKUP COPY
  - A BACKUP COPY OF INVARIANT CODE AND DATA IS KEPT.
  - OPERATING COPY IS COMPARED PERIODICALLY WITH BACKUP COPY.

APPLICATIONS IN PROGRAM VALIDATION

* UNDERSTANDING THE DYNAMIC BEHAVIOR OF THE PROGRAM HELPS GENERATION OF FURTHER TESTS
* SELF-METRIC SOFTWARE USEFUL FOR
  - EVALUATION OF EFFECTIVENESS OF A TEST
  - EVALUATION OF TESTEDNESS OF DIFFERENT PARTS OF A PROGRAM (PORTIONS OF PROGRAM THAT ARE EXECUTED MOST FREQUENTLY AND HAVE THE MOST EXTENSIVE SELF-CHECKING CAPABILITIES ARE MOST THOROUGHLY TESTED.)
  - MODIFICATION OF EXISTING TESTS TO EXERCISE UNTESTED PORTIONS
  - DETECTION OF ANOMALIES
  - GENERATION OF OUTPUT VALIDATION TESTS
  - PERFORMANCE EVALUATION AND OPTIMIZATION
* SELF-CHECKING SOFTWARE USEFUL FOR
  - TEST GENERATION BY FORMULATING ENTRANCE CONDITIONS OF EVERY SEGMENT OF CODE
  - DETECTION, LOCATION AND ISOLATION OF SOFTWARE ERRORS
  - TEST VALIDATION BY INSERTING ASSERTIONS

FUTURE RESEARCH

* EVALUATE THE COST-EFFECTIVENESS OF DIFFERENT SELF-METRIC AND SELF-CHECKING TECHNIQUES

* DEVELOP FORMAL METHODOLOGY IN DESIGNING SELF-METRIC AND SELF-CHECKING TECHNIQUES

* DEVELOP EFFECTIVE TECHNIQUES FOR GENERATING SELF-METRIC AND SELF-CHECKING SOFTWARE

* PROGRAM VALIDATION

* PROGRAM RESTRUCTURING FOR RELIABILITY, MAINTAINABILITY AND EFFICIENCY IMPROVEMENTS

INTEGRATED SOFTWARE DEVELOPMENT SYSTEM

Presented by:

Roy W. Coppinger

Higher Order Software, Inc.
843 Massachusetts Avenue
Cambridge, Mass. 02139
(617) 661-8900

INTRODUCTION*

The Integrated Software Development System (ISDS) is defined to be a comprehensive system structure that includes a basic set of principles, a basic set of tools and a basic set of standards for developing computer-based systems. The entire structure of ISDS is defined in terms of a common formalized methodology which underlies all elements of the system.[1] This formalized methodology supports the specification of the complete range of software development tools utilized within the structure. The principles of ISDS are applied to all phases of system development, and for all disciplines including design, verification, documentation, management, and maintenance. The axiomatic consistent development tools of ISDS include a specification language, a design analyzer, a structuring executive, automated documentation tools and advanced software management techniques. A specific goal of ISDS is to bring software development back in line with other engineering disciplines through the application of commonly understood and enforced engineering rules.

The system can be conceptually depicted as a management system operating on a data base with a sophisticated set of tools (Figure 1). The development data base contains the input and outputs of each ISDS process. The data management system uses the appropriate ISDS tool to verify the quality of a component of the data base or to analyze the interfaces between components.

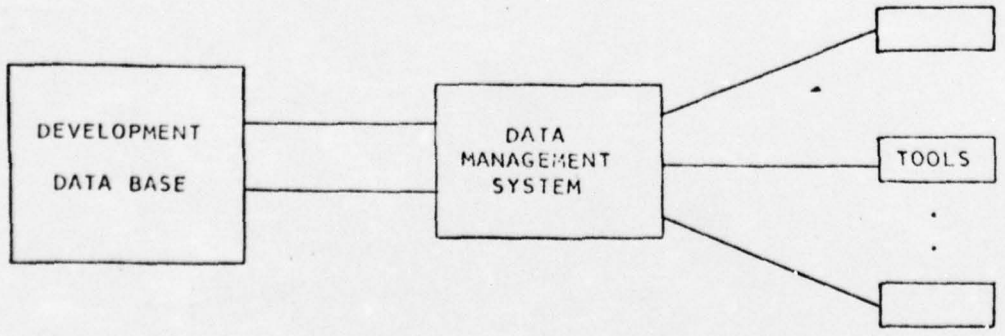


FIGURE 1: CONCEPTUAL ISDS SYSTEM

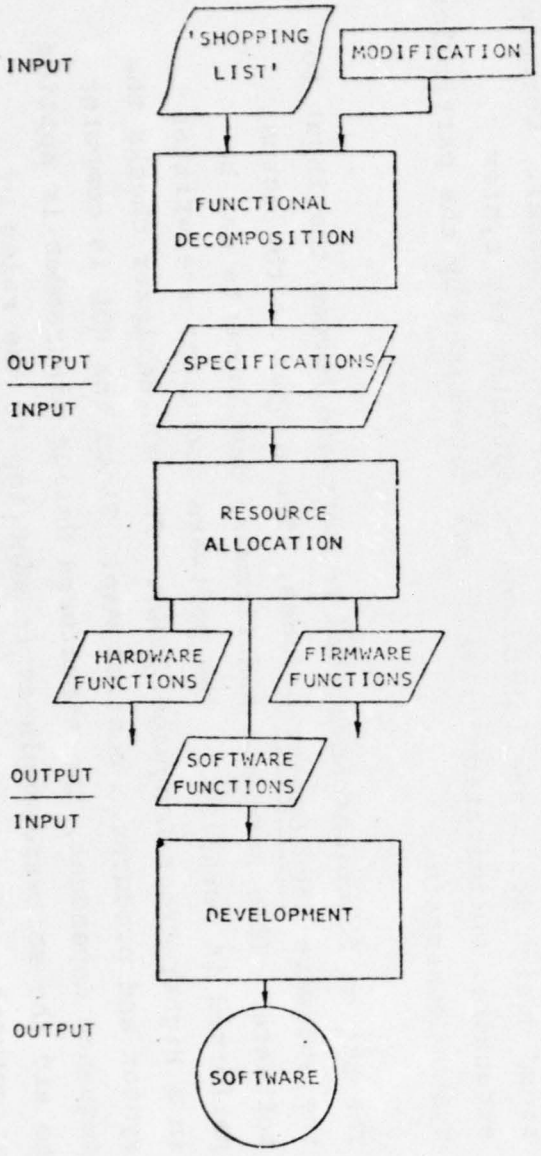
*Excerpt from Roy Coppinger's ISDS/HOS Conceptual Description Working Document, Chapter 3, Higher Order Software, Inc., in preparation, June 1976.

Each ISDS process will have identifiable inputs and outputs. Initially, the functional decomposition process has as input a set of requirements. These requirements may be in narrative form and can be a collection of 'desires' or a 'shopping list'. The output is a set of specifications. The specifications describe the input requirements as functions and functional relationships independent from a description of the execution properties of the system.

Formally, this process is aided by a specification language. The consistency of the specifications is verified by a design analyzer that checks the syntax of the specification languague statements, and produces a control map to graphically represent the tree-like structure of the system.

The set of specifications are input to the subsystem allocation process. The outputs are the specific function allocations for each part of the system (hardware, firmware, software). Computer aided design tools are introduced to optimize the system structure. Optimization criteria are determined by the particular system constraint.

The set of functions allocated to software become the input to the software development process. The output is the actual software. (The hardware and firmware developments can be performed in parallel.) The software functions are written in a Higher Order Language (HOL). The HOL analyzer checks the syntax and produces a control map. Since the HOL is computer sequence dependent, the structured design diagrammer is applied to aid the software engineer in adhering to the rules of structured programming. Figure 2 depicts the flow through the development cycle.



ISDS TOOLS

SPECIFICATION LANGUAGE

DESIGN ANALYZER

STRUCTURED FLOWCHARTER

DESIGN AID TOOL

DEVELOPMENT DOCUMENTATION

DEFINITIONAL INTERPRETER

STRUCTURING EXECUTIVE

FLOWCHART FOR SOFTWARE DEVELOPMENT

FIGURE 2

BACKGROUND

The Integrated Software Development System is based on the Higher Order Software[2] (HOS) methodology. HOS is a formal axiomatic system design methodology based on six axioms controlling interface correctness. HOS evolved from the APOLLO and SKYLAB projects. Figure 3 is a graphical representation of the software anomaly analysis for these projects.[3] Of particular interest are the preflight anomalies, 44% of which were discovered by eyeballing and 73% of which occured at software-software interfaces. The APOLLO on-board flight software effort was 2000 man-years, half of which was spent in verification. There were no software errors during flight but the achievement of this reliability was very expensive. Higher Order Software was developed as a formal methodology in order to reduce the costs of developing such highly reliable software systems.

Two major areas of concentration of the HOS methodology are interface correctness and static verification (automating the eyeballing process). Tools have been developed to automatically verify interfaces of programs without executing them. Interface correctness in HOS requires adhering to six axioms.

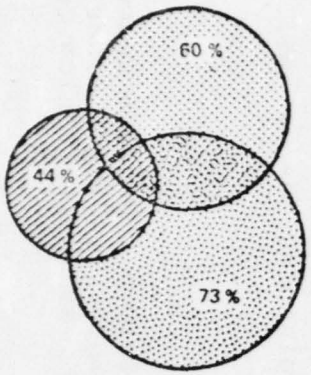
ILLUSTRATIONS OF THE AXIOMS OF ISDS*

For the purpose of illustration the fictional BRIGGEN system will be used as a framework for explaining the axioms.

---

*Excerpted from M. Hamilton, S., Zeldin, ISDS/HOS Conceptual Description Working Document, Chapter 4, Higher Order Software, Inc. June 1976. Under contract to U.S. Army Electronics Command, CENTACS, Ft. Monmouth, N.J.

A STUDY OF APOLLO AND SKYLAB SOFTWARE

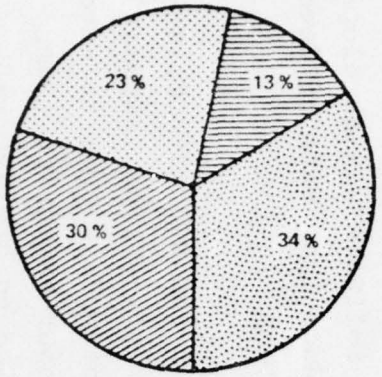


- OFFICIAL PRE-FLIGHT ANOMALIES (574)

found by eyeballing

existed in previous releases

caused by software-software interface problems



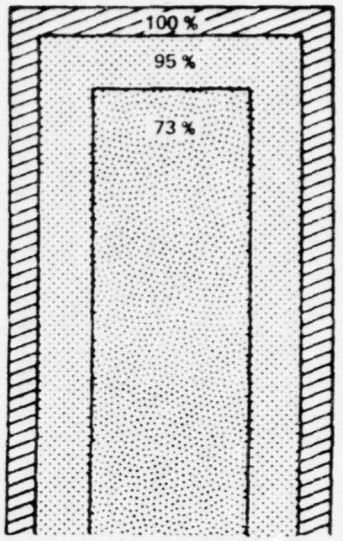
- OFFICIAL PRE-FLIGHT ANOMALIES (574)

would have been prevented by improved programming style

would have been prevented by design techniques

would have been prevented by techniques of building software

no prevention found yet



- RECORDED REAL-TIME SYSTEMS PROBLEMS (42) DURING APOLLO FLIGHTS CAUSED BY:

manual development processes

system interface design philosophy

real-time human error

pure software errors during flight: NONE

FIGURE 3

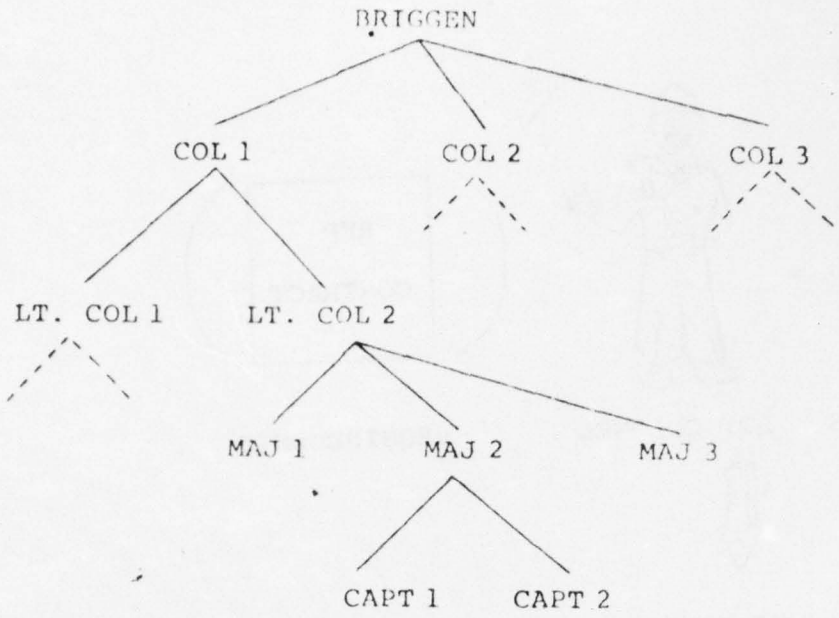


FIGURE 4

BRIGGEN Invocation Tree*

The management hierarchy shown in Figure 4 is often referred to as a tree. BRIGGEN, COL 1, and CAPT 2 have a particular position of responsibility. We often refer to this geographic position as a "node". When we refer to COL 1 giving orders to LT. COL 1 and LT. COL 2, COL 1 is referred to as a "controller" or as a "module". When COL 1 receives orders from BRIGGEN he is referred to as a function. A module is a supervisor; a function is a subordinate. A function receives its input values in the input variables, performs its operation, and placed the output values in the output variables.

The six axioms of ISDS/HOS are explained in Figures 5 through 10 using the BRIGGEN system.

*An invocation tree is a representation of a control map which contains only function names.

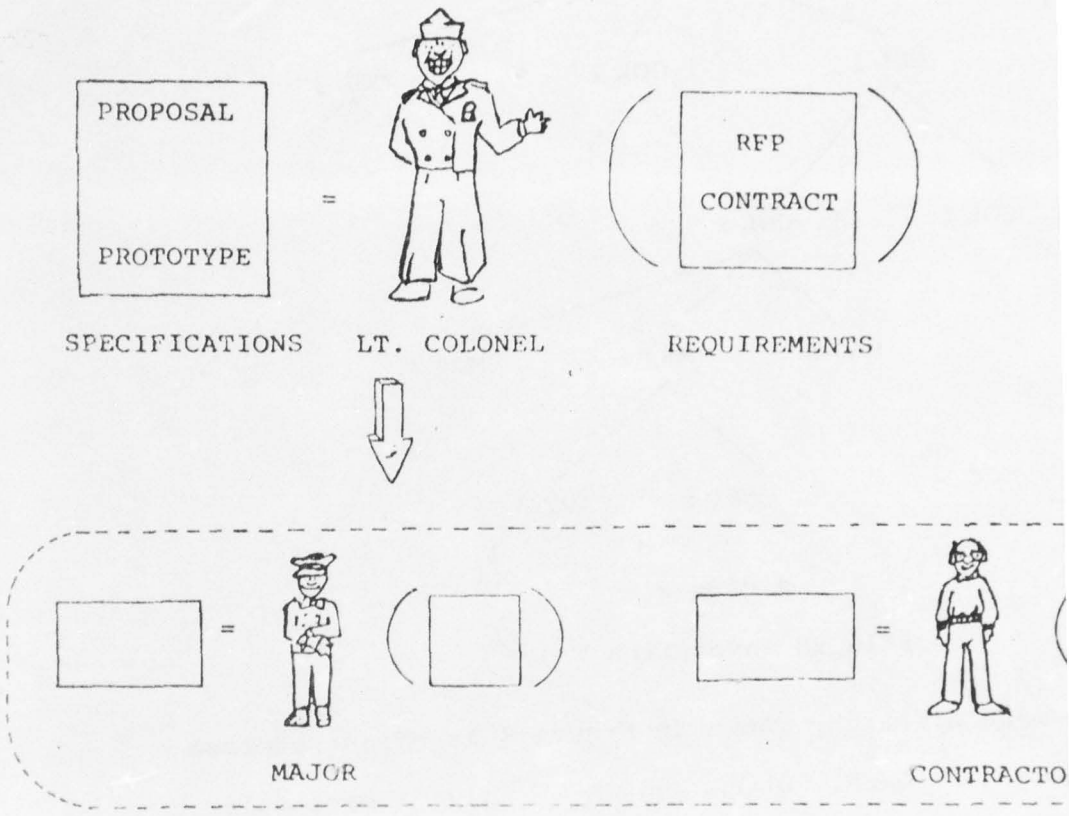


FIGURE 5 AXIOM (1) INVOCATION RIGHTS

AXIOM 1

Axiom 1 is illustrated in Figure 5. The LT.COLONEL is o
on REQUIREMENTS to produce SPECIFICATIONS.

SPECIFICATIONS = LT.COLONEL (REQUIREMENTS)

If the MAJOR and CONTRACTOR are the only functions tha
to a control level controlled by LT.COLONEL, then both
and CONTRACTOR contribute to completing the LT.COLONEL
We refer to this contribution as invocation. Axiom 1
each invocation to the total problem defined by LT.COL
LT. COLONEL controls only the invocation of MAJOR and

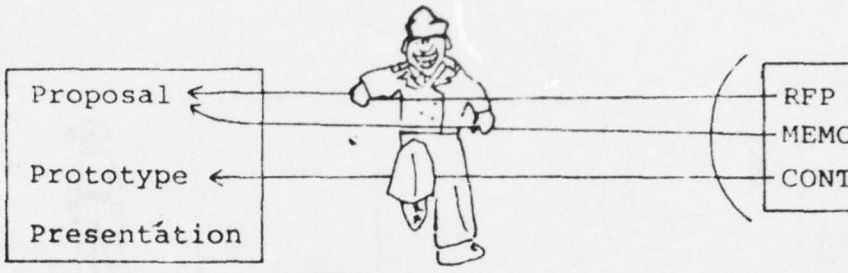


FIGURE 6 AXIOM (2) RESPONSIBILITY RIGHTS

AXIOM 2

In Figure 6 LT.COLONEL is in charge of taking the inp
values of REQUIREMENTS; RFP, MEMO, and CONTRACT and p
output values of SPECIFICATIONS; PROPOSAL AND PROTOTY
refer to this input/output relationship as responsibi
Axiom 2 relates each input value so as to control tha
sibility in such a way that only one output value is
with a particular input value and for every input val
must be a corresponding output value.

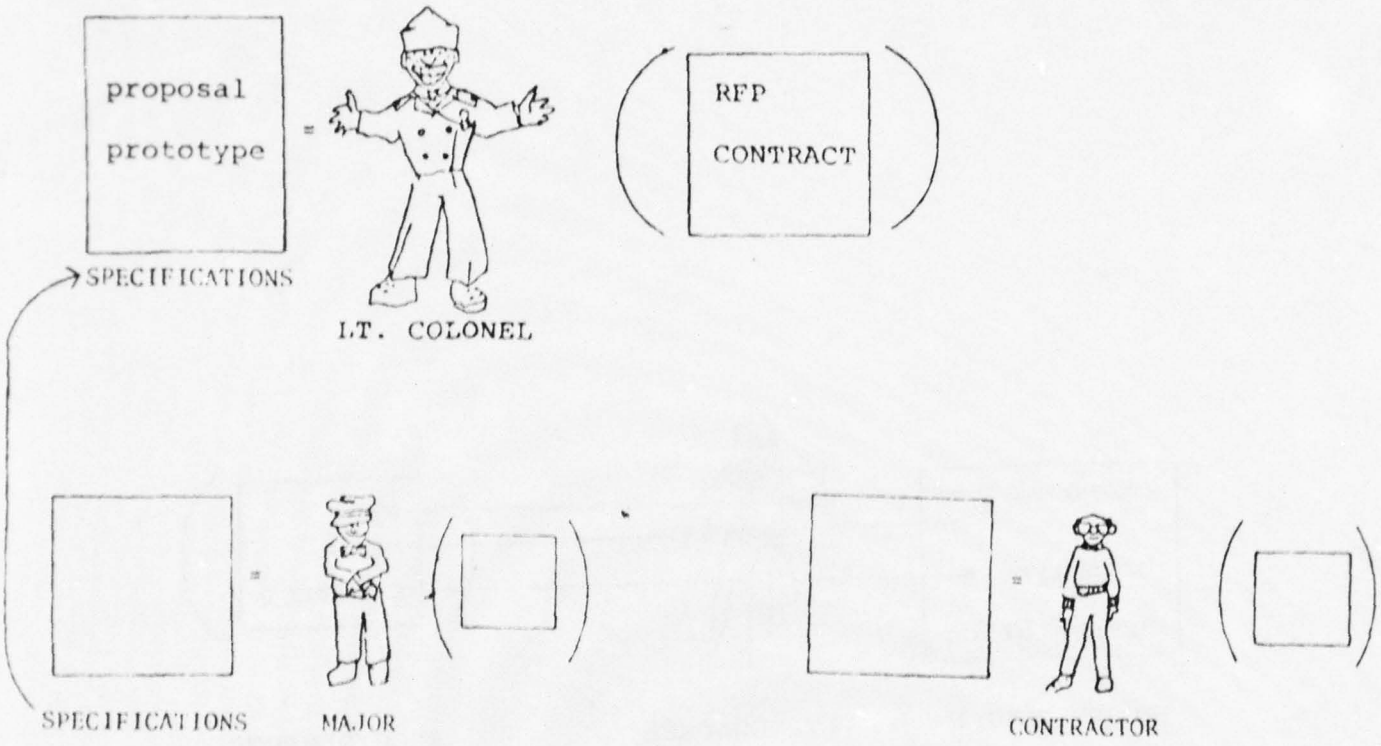


FIGURE 7 AXIOM (3) OUTPUT ACCESS RIGHTS

AXIOM 3

The manner in which a module can locate a value of its input or output variables is via access rights. We distinguish input access rights from output access rights. Axiom 3 defines a module's control of output access rights for the functions it controls.

In Figure 7 LT.COLONEL has output access rights to the variable SPECIFICATIONS. We can locate an element of the output set such as PROPOSAL or PROTOTYPE by accessing SPECIFICATIONS. When we refer to control of output access, we imply that a variable in the set that is under output access rights for LT.COLONEL, appears as a variable in the set under output access rights for MAJOR or CONTRACTOR.

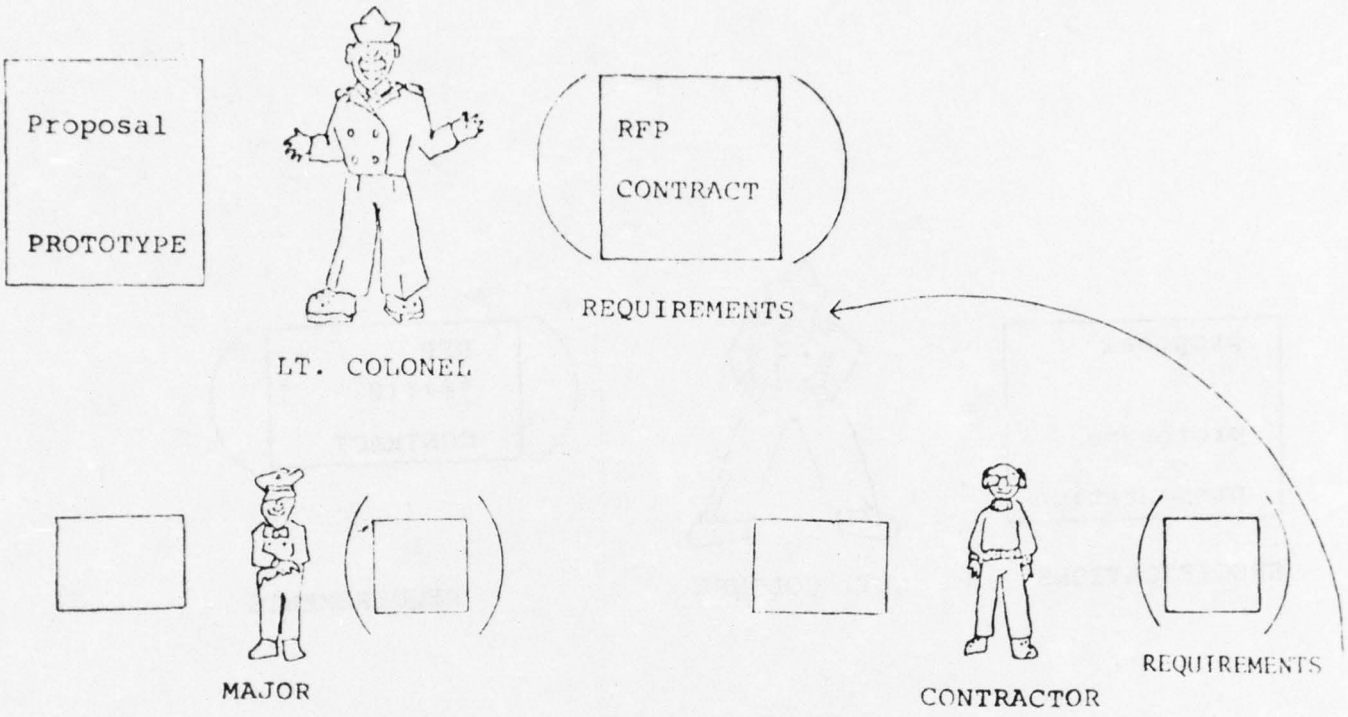


FIGURE 8 AXIOM (4) INPUT ACCESS RIGHTS

AXIOM 4

Axiom 4 defines a module's control of input access rights for the functions it controls. In Figure 8 LT.COLONEL has input access rights to REQUIREMENTS. We can locate an element of the input set of LT.COLONEL (such as RFP or CONTRACT) via these rights.

We refer to control of input access when we imply that a variable in the set that is under input access rights for LT.COLONEL, also appears as a variable in the set under input access rights for MAJOR or CONTRACTOR.



FIGURE 9 AXIOM (5) REJECTION RIGHTS

AXIOM 5

Techniques used to determine if input values are valid are controlled by the module itself. This means that if LT. COLONEL knows he can accept only RFP or CONTRACT, then he can reject anything else. He can do this without requesting his subordinates to do it for him. This type of control (i.e., rejection control), limited to the module itself, is referred to as Axiom 5 and is illustrated in Figure 9.

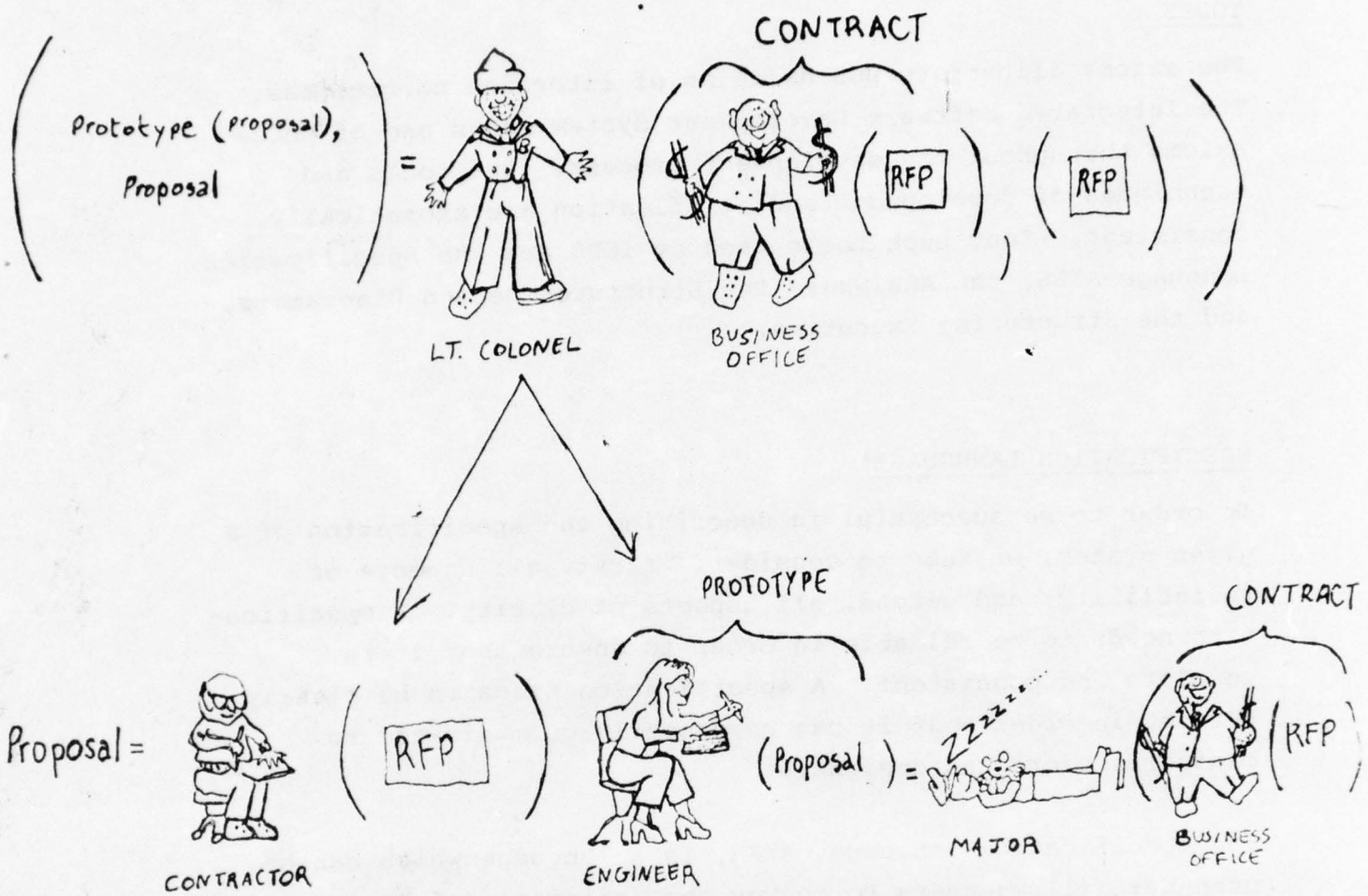


FIGURE 10 AXIOM 6 ORDERING RIGHTS

AXIOM 6

Axiom 6 relates the ordering of functions with respect to their controller. Axiom 6 requires that LT. COLONEL be responsible for the ordering of functions MAJOR and CONTRACTOR. In the example of Figure 10, since PROTOTYPE requires information from PROPOSAL, LT. COLONEL would insure that CONTRACTOR produced PROPOSAL before MAJOR was invoked to produce PROTOTYPE.

TOOLS

The axioms illustrate HOS handling of interface correctness. The Integrated Software Development System makes use of these axioms throughout the development process. The tools and techniques of development and verification are axomatically consistent. Four such tools used by ISDS are the Specification Language-AXES, the Analyzer, the Structured Design Diagrammer, and the Structuring Executive.

SPECIFICATION LANUGUAGE*

In order to be successful in describing the specification of a given system, we need to consider: first, all aspects of reliablility; and second, all aspects of clarity. A specification needs to be reliable in order to ensure that it is complete and consistent. A specification needs to be clearly defined in order that it can be properly communicated to managers, users and designers.

The specification language, AXES, is a language which can be used by: (1) managers to convey the definition of a given system; (2) users to define a specification for a given system; (3) designers to create mechanisms which are buliding blocks for a given specification. AXES is intended to provide the necessary mechanisms for the specification of any given system. In addition, it will include recommended building blocks for the specifications of representative systems. From the language mechanisms, new building blocks can be defined. In addition, other new building blocks can be constructed from existing building blocks for individual system needs. From these building blocks, systems can be described as abstract or as detailed as desired.

---

*Excerpt from The Foundations for AXES: A Specification Based on Completeness of Control, Charles Stark Draper Laboratory, Inc., March 1976. Under contract to U.S. Navy, Naval Electronics Laboratory Center, San Diego, CA.

To date, extensible features in systems, such as programming languages, have not been generally successful[4,5]. We believe that the mechanisms provided in the specification language will facilitate the development of extensible systems as first envisioned.

AXES will provide the mechanisms for stating the properties of a given system. Several properties of specification distinguish it from the implementation of that specification. Many times, when designers specify a given system, they assume implicitly that the implementations they are familiar with account for those properties. For example, just because one implementation system provided a lock mechanism for shared data does not mean that all implementation systems will do so. It is very difficult for designers to remove these biases, especially when the assumptions made are accounted for by all known implementation systems to date. We should consider, however, that some implementation features that are common today may not be the type of implementation features that we would envision for the future. More significantly, they may be restricting us in the design process more than we realize.

A given specification should be independent of the particular "machine" that is used to implement that specification. That is, the specification is not dependent on implementation tools such as hardware, software, a human operator, compiled code from a higher order language, an interpreter, or a software subroutine. Each specification assumes only that there is a machine mechanism which will realize the properties of that specification. This distinction in design is important; for once a system is designed dependent on implemenation tools, the independence or portability of that specification becomes limited.

A given specification should be independent of the particular system design concepts that are assumed in the implementation of that specification. For example, for a given specification, functions can communicate, in concept, cyclically with each other. However, the implementation of that specification could assume an asynchronous environment whereas another implementation could assume a synchronous environment. It is advantageous to define a specification independent of the system design concepts of the implementation in order that a given specification could be transferred from one implementation to another implementation.

A given specification should be defined with unabiguous symbols, words or combinations thereof. With languages, such as English, there is much confusion as to intent or meaning; since more than one meaning can often be attached to the same word when used in one particular context; or the same word can often have different meanings depending on the context in which it is used. (In fact, change of inflection in one's voice can change the meaning of words.) It is entirely possible to design a formal language in which the phrases look like English[6]. In AXES, we can show the relationship between the formal semantics of an abstract control structure and its English-like equivalent. But we must also consider the syntax associated with that control structure to be unambiguous.

A successful specification must, therefore, be designed independent of implicit assumptions; it must be designed independent of its implementation tools; it must be designed independent of system implementation design concepts; it must convey intent unambiguously; and it must provide mechanisms to describe the properties of systems in as abstract a manner as desired.

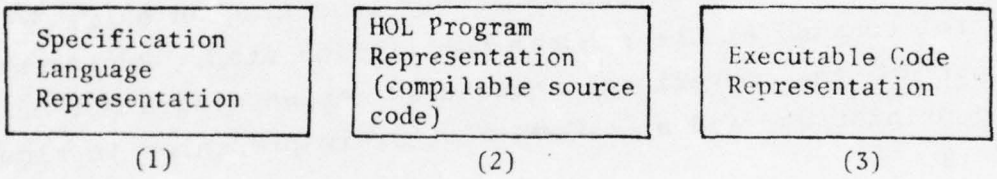
AXES will provide a reliable means in which to define a successful specification. Reliability is to be obtained from the properties of systems defined by Higher Order Software[1]. These properties are embedded in the specification techniques for defining

abstract control structures. Each abstract control structure uses abstract data types to complete a given system specification.

With AXES, the abstract control structures can be defined in such a way as to both conform to the formalism required for reliability and to convey intent of the designer to managers and users of a given system application.

ANALYZER

The main function of an analyzer is to guarantee that a given hierarchical system is consistent with the axioms. There are at least representations of a software problem:



An analyzer addresses reliability issues in the first two representations, specifically, "do representations 1 and 2 violate the six HOS axioms?" and "is represention 2 in compliance with the specification in 1?".

The most costly errors in software development are those that occur in the design process and remain undiscovered until system testing. By having analyzers available for the specifications and the HOL code the Integrated Software Development System can eliminate a large percentage of the errors before the code is executed.

A Parnas type specification of a Navy Message Processor was manually analyzed revealing 29 errors in the specification. Twenty-seven of the 29 errors were found to be interface errors that would be detected by the analyzer[7].

STRUCTURED DESIGN DIAGRAMMER

The Structured Design Diagrammer[8,9,10] (also referred to as the Automated Flowcharter) is a computer program whose function is to produce documentation from source statements for an HOL. Up to date flow diagrams can be produced automatically for each revision of a program module. These flow diagrams aid the programmer/engineer to use structured programming conventions by providing a visual representation of the block structure, the scope, and the data flow inherent in the program.

A Design Diagrammer has been developed and implemented for HAL, a block structured Higher Order Language (HOL). This design diagrammer is used routinely as a production tool for software development for the SPACE SHUTTLE. Programmers are able to obtain this output by simply inserting an extra job control command at the program compilation step. Representative output of the operational structured design diagrammer on a line printer and and a CalComp plotter is presented in Figures 11 and 12.

Demonstrated advantages of the existing Design Diagrammer include:

1. Automation of the documentation process.

2. Improved management control of software development.

3. Effective enforcement of structured programming techniques.

Use of automated documentation in the SPACE SHUTTLE effort has materially reduced the cost of flow diagram preparation and has resulted in rapid communication of up-to-date documentation among all participating organizations.

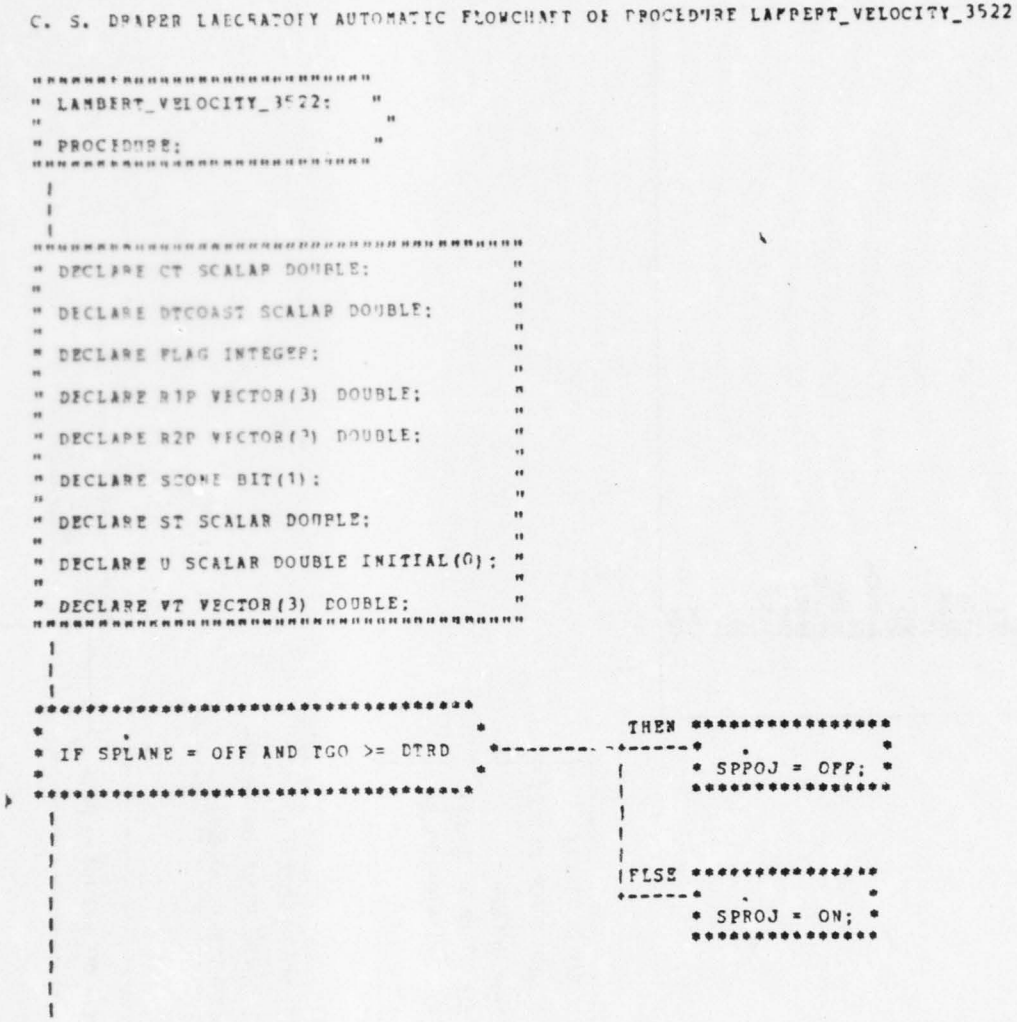


FIGURE 11: EXAMPLE OF DESIGN DIAGRAM IN PRINTED FORM
(From Space Shuttle Guidance Routine)

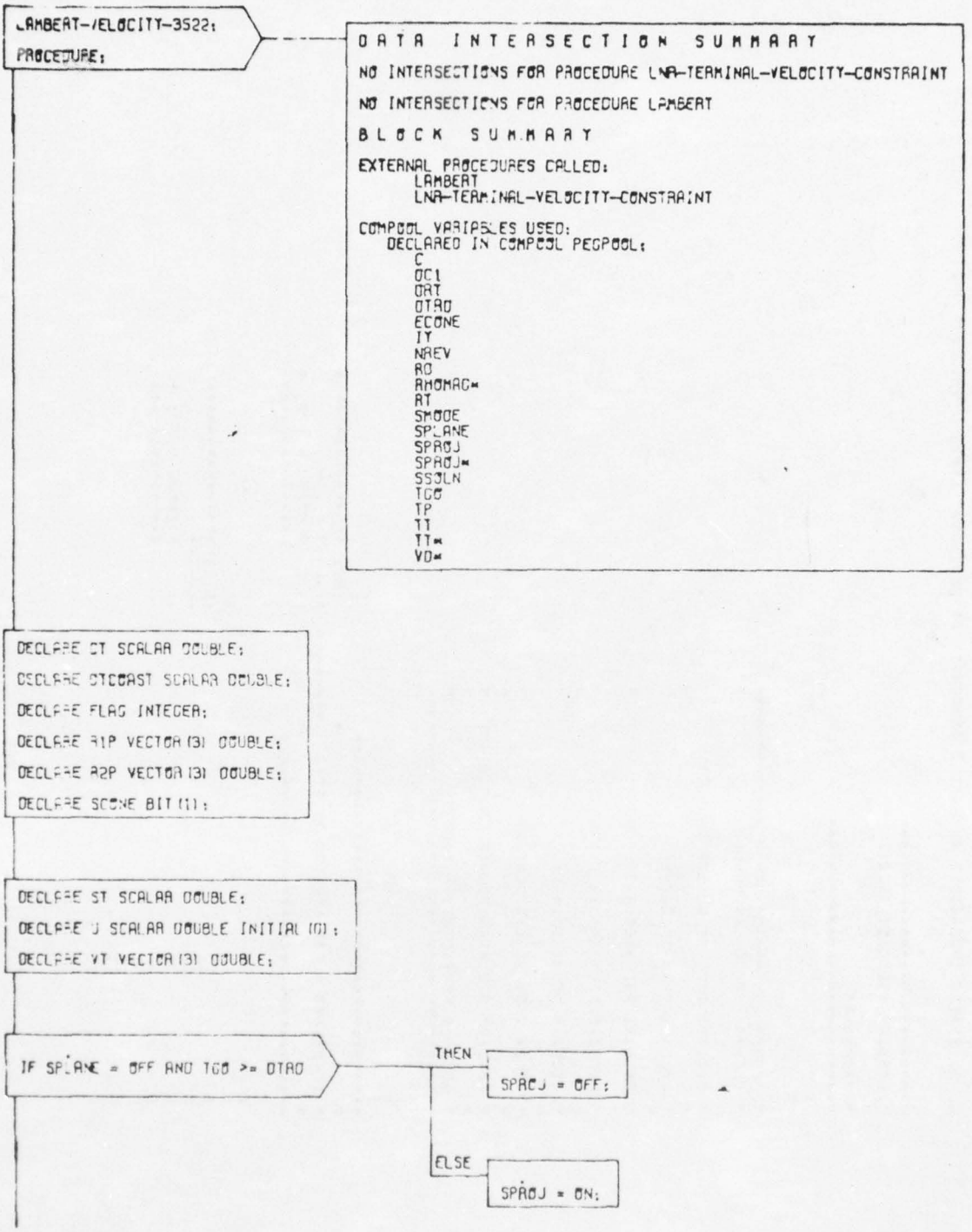
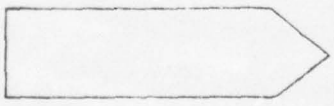


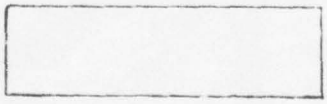
FIGURE 12: EXAMPLE OF DESIGN DIAGRAM IN PLOTTED FORM
(From Space Shuttle Guidance Routine)

Prior to the automation of the structured design diagrams, programmers and engineers produced design diagrams manually for the purpose of structuring an algorithm before it was coded. Now, the automatically produced design diagram is used as a comparison with the original manually produced design and as a means of producing up-to-date and automatic documentation of the computer program.

The salient features of the structured design diagrams are more applicable for software development than conventional flow diagrams. Figure 13 illustrates the two conventions used in generating design diagrams. Referring back to Figures 11 and 12. Lines connect nested decision levels and linear execution flow. Notice that after each logical step has been completed control returns to the main flow of the algorithm. It is easier to see the state of the system at each node. The nested decision levels are clearly distinguished. Given the algorithm in this form, the equation can be "eyeballed" for correct formulation.



Used for all relations and Boolean expressions.



Used for invocations (explicit and implicit).

FIGURE 13: DESIGN DIAGRAM CONVENTIONS

STRUCTURING EXECUTIVE

The Structuring Executive[1] is a lower virtual layer module with respect to a given hierarchical HOS system in its dyanmic state. The structuring executive is used to implement, in real time, multiprogramming control constructs. The structuring executive:

1) controls the ordering of those modules which can vary in real time dependent on operator selection.

AD-A033 625 ROME AIR DEVELOPMENT CENTER GRIFFISS AFB N Y F/G 9/2
PRESENTATIONS AT THE RADC/AR?A INVITATIONAL DOD/INDUSTRY CONFER--ETC(U)
AUG 76

UNCLASSIFIED NL

4 OF 5
AD
A033625



2) assigns priorities to processes based on their relative priority relationships (Axiom 6) for each control level.

3) prevents a violation of the HOS axioms so that no two processes can conflict with each other.

4) determines when the total resources of the computer are approached.

The structuring executive is the dynamic analog to the analyzer's static verification of interfaces.

CONCLUSION

The lack of disciplined development system is the major contributing factor to the Department of Defense's problem of producing reliable software[11, 12]. The Integrated Software Development System will contain the tools, techniques, and principles necessary to produce reliable software at a cost-effective price.

REFERENCES

1. Hamilton, M., and Zeldin, S., "Higher Order Software - A Methodology for Defining Software," IEEE Transactions on Software Engineering, Vol. SE-2, No. 1, March 1976.

2. Hamiltion, M., "Design of the GN&C Flight Software Specification," Charles Stark Draper Laboratory, Cambridge, MA, Document C-3899, February 1973.

3. Hamilton, M., and Zeldin, S., "Higher Order Software Techniques Applied to a Space Shuttle Prototype Program," in *Lecture Notes in Computer Science*, vol. 19, Goos and J. Hartmanis, Ed. New York: Springer-Verlag, pp. 17-31, presented at Program. Symp. Proc., Colloque sur la Programmation, Paris, France, April 9-11, 1974.

4. Standish, Thomas A., "Extensibility in Programming Language Design, " SIGPLAN Notices, Volume 10, July 1975.

5. Christenson, C., and Shaw, C. J., ed., "Proceedings of the Extensible Languages Symposium, SIGPLAN Notices, August 1969.

6. Leavenworth, Burt M., and Sammet, J. E., "An Overview of Nonprocedural Languages", SIGPLAN Notices, Volume 9, No. 4, April 1974.

7. Pankiewicz, R., "Message Processor Specification," Specification Language Memo 8, Charles Stark Draper Laboratory, Cambridge, MA December 11, 1975.

8. Hamiltion, M., and Zeldin, S., "Top-Down/Bottom-Up, Structured Programming and Program Structuring," Charles Stark Draper Laboratory, Cambridge, MA, Document E-2793, August 1973.

9. Daly, W., "Automatic Flowcharts," Draper Lab. MERCURY Memo No. 53, March 1974.

10. Goddard, G., "Users' Guide to the Automatic Structured Flowcharter," Draper Lab. Document No. C-4632, April 1976.

References continued:

11. Kossiakoff, A., Sleight, T. P., Prettyman, E. C., Park, J. M., and Hazan, P. L., "DoD Weapon Systems Software Management Study," Johns Hopkins Univeristy Applied Physics Laboratory, APL/JHU SR 75-3, June 1975.

12. Asch, A., Kelliher, D. W., Locher III, J. P., Connors, T., "DoD Weapon Systems Software Acquisition and Management Study Volume 1, MITRE Findings and Recommendations," Vol. 1, MTR-6908, May 1975.

VERIFICATION AND ERROR DETECTION USING A SYMBOLIC EVALUATOR

T. E. Cheatham, Jr.
Harvard University
Center for Research in Computing Technology
Cambridge, Massachusetts 02138

(Notes on a talk delivered on 5 August 1976 at the Invitational DOD Industry Conference on Software Verification and Validation, Syracuse, New York.)

1. Overview

- The Harvard Program Manipulation System (PMS) - a collection of tools for aiding in the development, validation, and maintenance of programs.
- Some remarks on the programming process.
- The EL1 Symbolic Evaluator.
- Application of the Symbolic Evaluator to program verification and fault finding.

2. The Harvard Program Manipulation System

- A comprehensive programming system with a variety of tools and facilities for aiding in all aspects of the process of program development, validation, and maintenance.
- Some tools and facilities:
  * The EL1 programming language.
  * Rewrite facilities and editors.
  * Probe and monitor facilities.
  * Program checker.
  * Program verifier.
  * Program data base and query facility.
  * Symbolic Evaluator.
  * Cost expression development tools.
  * High level optimizers.

3. Remember the Programming Process

The "programming process" commences with some "abstract program" -

- sometimes this is stated mathematically, e.g. sorting, parsing solving linear equations.

- but more often not
  payroll, inventory, MIS, satellite tracking,...

But, in any event, the abstract program is "understood" by humans to constitute a specific algorithm at some level of abstraction.

Given the initial "abstract program", the programming process involves making choices:

- division of the program in to modules, sub-modules, and so on

- representation of data objects

- representation of operations (transactions, transformations, etc.)

This may (or should) be done at several levels - consistent with the tenets of Structural Programming.

The goals of the choice are

1. preservation of functionality

2. efficiency

3. preservation of clarity

The domain of choice is some inventory of possibilities; these are strongly dependent upon

- the language and system, and

- the experience and skill of the programmer.

The bases for makign a choice are facts or predicates which enable a given choice. How do we get these facts?

- inherent - i.e. we must maintain functionality

- deduced - e.g., we note that some array is sorted at some point and take advantage of that fact

- decided arbitrarily - e.g., we decide that an identifier will be less than 30 characters long, or we decide that some array will be sorted at some point.

We propose a strong need for dramatic departure from the current programming practice - that is, these facts are every bit as much a part of the find "program" as is, for example, some load accumulator instruction occuring in a run time load module.

A dual problem to that of making choices is that of optimization of some extant program

- doing x in a "better" way
- not doing some thing because we know we do not need to

These are the "usual" kinds of optimization

- common subexpression elimination
- register allocation
- retrieval of storage from "dead" variables
- and so on.

There are also a number of "high level" optimizations which might be considered.

- recursion removal
- loop fusion
- loop cleavage
- program simplification
- loop reduction
- and so on.

But again, optimizing transformations are enabled by facts (predicates) about the program.

The above requires reasoning about a program - and that in turn requires a calculus, a collection of mathematical functions and rules which permit us to describe and reason about what is happening in the execution of a program.

- these must obviously include the usual arithmetic and logical operations
- they must also include a logical calculus (e.g., the first order predicate calculus)
- they may also include such mathematical operations as finite sums and products, functions which define arrays, and list structure (e.g., that array whose j^{th} element is f(j), and the least j such that..., and so on.

The EL1 Symbolic Evaluator

The EL Programming Language is a complete programming language which has been used in the development of several systems programs

- integers, reals, characters, booleans.
- homogeneous arrays
- records (structures or non-homogeneous arrays)
- pointers
- shared values
- block structure with local variables
- loops
- dynamic scoping
- and so on.

Given same program, the goal of the Symbolic Evaluator is to develop functions which describe the values of all variables in all context - where the values of unknown quantitatives are represented symbolically

The SE has three major components

- symbolic execution
- loop analysis
- simplification (including a theorem prover)

Symbolic Execution

The symbolic execution of some program (represented as, essentially, list structure) results in

- a "shadow" of the program - a representation in which modes of all quantities are known, all implicit computations are made explicit (e.g., dereferecing, the "hidden" computation involving loop variables, and so on), and manifest constants are propagated.
- a context graph, or flow graph representing the possible paths of control.
- an environment, consisting of a "location" for each distinct variable occuring in the program, plus a list of the values of that variable in various contexts (represented as symbolic expession, in general)

Loop Analysis

The loop analyzer does two things

- *determines the value of each variable which may be modified within the body of the loop as a function of a symbolic quantity, k, which denotes the cycle index (i.e., k would be one on the first cycle; two on the second, and so on).*

- *solves for the value k_L, the value of the cycle index on the last cycle taken, in general a symbolic expression.*

The basic method is as follows: We postulate, for each quantity which may change in the body of the loop, some symbolic value, B, as its value at the beginning of a general (the kth) cycle. We then do a symbolic execution of the loop body. Assuming we are to cycle again, a given variable then has some symbolic value, A, at the end of the kth (beginning of the k+1st) cycle. The quantities B and A provide a recurrence relation for the change in the variable during the kth cycle, with the value of that variable immediately preceeding the loop as boundary condition.

Solving the recurrence relations is straight forward if the differnce A-B happens to be constant or a function of k.

In general, of course, the difference A-B is conditional. If this is the case we can look for standard kinds of conditionality and still often obtain a function describing the value of the variable at the beginning of a general cycle

- *one example is "first/rest" behavior which happens when for some variable, the loop has two stable states*

- *another example is "subset" behavior when a variable is modified on same subset of the cycles*

- *and, so on.*

The important point is that we can study conditionality by postulating, for example, first/rest behavior, and then verifying that this is or is not the case by substitution of appropriate values into this symbolic expression describing the values of variables at the end of a general cycle in order to test the hypothesis.

If all else fails, we can always represent the value of a variable by a recursion equation - byt even these are subject to reasoning analysis, and simplification.

The number of cycles taken, k_L, is determined by finding for each exit (say the ith) that predicate involving k which enables that exit, say $p_i(k)$. k_L is then the least j such that $p_1(j)$ or $p_2(j)$ or..., a function which often simplifies to a constant or symbolic values (e.g., the length of same array in same dimension).

Simplification

A powerful simplifier is required to keep the size of expressions in hand and to render them fit for human consumption.

At present the boolean expression simplifier includes a complete linear solver for systems of linear inequalities.

The simplifier is divided into two components, one is a "black box" simplifier which handles standard cases (e.g., 1+1=2, TRUE or P=TRUE etc.) and puts expressions into a normal form. The other is, essentially, a collection of axioms which can be applied to an expression. New axioms are readily added by the user in a "human" notation.

4. Applications of the Symbolic Evaluator

Given some programs P and the result of its symbolic evaluation, we envision several applications of information derived about the program

- *high level transformations to improve a program.*
- *debugging assistance*
- *automatic or semi-automatic selection of data representations in certain studard cases (e.g., sets, sparse arrays, and the like)*
- *program verification*
- *mechanical error detection.*

We look at the latter two in somewhat more detail.

Verification

The use of a full symbolic evaluator in program verification often precludes the need to insert inductive assertions inside a loop - the expressions describing the behavior of a variable on a general cycle and at exit are often sufficient. For example the proof that the usual gcd algorithms are correct is a simple theorem concerning the exit values of affected variables.

The "frame" problem is often significantly allocated. For example it may be manifest that the value of some quantity does not change or that the values in some array at the end of a loop are the same set of values as were in the array at the beginning - eliminating the need to set up and prove the verification conditions insuring this for every operation within the loop.

Mechanical Error Detection

By error detection we refer to insuring that a program does not do certain things it should not do - independent of whether it does what it should do. Examples are

- *subscript range verification*
- *NIL pointer selection*

- non-initialized variables
- excluding division by zero
- and so on.

In addition, we expect to extend the fault finding mechanisms to guarantee program protection and security. That is, if a certain class of transactions with certain variables and/or procedures is denied the techniques of fault finding can be applied to guarantee that these restrictions are not violated.

5. Where Are We?

At the present time, the Symbolic Evaluator is " limping" - by late fall it should be operational but lacking in certain sophistication.

We have commenced studying the application of the SE to program verification and fault finding.

A SYSTEM FOR THE VERIFICATION OF JOCIT PROGRAMS

Bernard Elspas and Jay M. Spitzen
Stanford Research Institute
Menlo Park, California

SUMMARY

This paper describes current research and development** in the Computer Science Group at SRI aimed at ultimately producing a programming environment in which a JOVIAL programmer can write, debug, and prove correctness for his programs. The first phase (RPE/1) of this work resulted in an experimental system at SRI capable of handling verification of small program modules written in a subset of JOVIAL (mainly of J3). The current (RPE/2) phase is aimed at extending the capabilities of the original system to handle verification for programs that use most of the the features of JOVIAL (JOCIT-J3). Moreover, the system is to be transportable to the RADC-MULTICS Honeywell 6180 computer.

Both phases of this work have been based on the technique of specifying program behavior by inductive assertions and entry/exit assertions (the Floyd approach). The verification systems produce verification conditions (VCs) automatically for any legal program that has been properly annotated with inductive assertions. Each loop of the program program must be interdicted by at least one such assertion. These VCs are mathematical-logical formulas which, if they can be validated, serve to demonstrate consistency between the program and user-supplied assertions. A user-guided deductive subsystem of the verifier is used to generate formal mathematical proofs of the VCs.

The front end of the verification system consists of (1) a parser transducer that analyzes a JOVIAL program file and transduces it to tree-structured form, and (2) a verification condition generator (VCG) that that produces VCs from this parse. Thus, the front end can be regarded as a kind of 'compiler' (a verification condition compiler) that performs some syntactic checks on the input program, but which outputs VCs instead of compiled code. The actual semantic checking of the program is, of of course, carried out on the VCs by the deductive system. The VCG operates by computing the "weakest precondition" WP[S,Q] for each segment S of the program as determined by the semantics of JOVIAL and the assertion Q affixed to the end of the segment. Thus, WP[S,Q] is a formula in predicate calculus, which if it is valid on entry to the program segment S, requires that Q be valid if execution reaches Q. In Hoare's notation:

$$WP[S,Q]\{S\}Q \text{ holds.}$$

**Under Contract F30602-76-C-0204 with Rome Air Development Center. The first phase (also for RADC) was conducted under Contract F30602-75-C-0042.

Moreover, WP[S,Q] is the weakest such formula, i.e., it is implied by any formula P with the property P{S}Q. The semantics of JOVIAL are incorporated in (Hoare-type) inference rules -- one for each JOVIAL construct -- that specify the mapping WP: S x Q --> Q'. For example, if S is a JOCIT simple variable assignment, var = expr $, the corresponding transduced form is (:= var expr), and WP[S,Q] is given by:

(SUBST expr var Q)

where SUBST denotes the substitution of (a free occurrence of) expr for each (free) occurrence of var in Q. Similar formulas apply for each of the JOVIAL statement constructs. When the VCG encounters an assertion at the head of a segment S in this process of working backwards through the program, VCG appends the new verification condition for that segment to its list of previously collected VCs. Finally, when the input assertion at the head of the whole program is reached, VCG constructs an overall implication of the form:

<input-assertion> ==> <previously-constructed-formulas>

and outputs this formula.

The VCG also analyzes all of the declarations in the (transduced) subject program and creates subsidiary assertions from these declarations. In particular, this permits VCG to carry out a hierarchical treatment of of subprocedures and functions invoked within a JOVIAL program, so that that the proof of correctness of the main program can be decoupled from the proofs for its submodules. The proof partitioning process is an important aspect of program verification as carried out by our system, since it makes manageable the complexity associated with larger programs, provided they are written in structured form. It has the benefit that subprocedures do not have to be reanalyzed (and proved correct) for each invocation in the main program. However, the human verifier must supply entry- and exit assertions that adequately characterize each such subprogram.

The heart of this program verifier, as with other such systems, is its deductive component, or 'theorem prover'. The purpose of the theorem prover is to attempt proofs of validity for the mathematical-logical formulas (VCs) constructed by the VCG. It should be noted that the VCG eliminates all of the details of control semantics of the subject JOVIAL program in the process of constructing the VCs. The VCs are formulas (in first-order predicate calculus) which a human mathematician could usually decide as to validity without any knowledge of JOVIAL programming, if he were tireless enough. Most of the facts needed in proving validity of VCs are purely mathematical ones, residing in the domains of the algebra of integers and real numbers, and of propositional logic. Such facts as commutative, associative and distributive laws, the special properties of 0 and 1, the properties of equality- and inequality relations, and the definitions of a few other important primitive functions (exponentiation, log, antilog and square root, for example) are built into the deductive

system. A deductive subsystem called the 'expression simplifier' plays a major role in such simple deductions. The deductive system is also informed about methods for handling proofs of logical formulas built up from the connectives, AND, OR, NOT, IF, IMPLIES, and the logical quantifiers, "forall" and "exists", although the latter play a relatively minor role. The logical formula domain of the VCs is here referred to as the (abstract) 'assertion language' of the verifier. The term 'concrete' will be used to refer to expressions in their original untransduced form. Any assertion language formula can legally appear as an entry/exit assertion or as an inductive assertion in a subject program. The (concrete) assertion language contains all legal, executable Boolean formulas of JOVIAL, and also allows logically quantified Boolean formulas and certain other modes of expression.

The Deductive System comprises several distinct layers with different capabilities and modes of user interaction. A Proof Supervisory Executive subsystem permits the user to invoke deduction on any selected VC (or portion thereof). This Proof Supervisor is an implementation of the Smullyan method of analytic tableaux. It first attempts purely automatic deduction, and often this is all that is needed. When automatic deduction is unsuccessful, the tableaux system asks the user for guidance, while displaying and saving the state of the partially successful proof. The user can then invoke additional information in the form of lemmas, definitions, or previously proved results, and this information becomes part of the hypothesis of the formula to be proved. The tableaux system also permits the user to invoke special 'black boxes' for certain kinds of deductions. One such special deduction subsystem is a decision mechanism for (an extension of) Presburger arithmetic. This 'Presburger box' can decide validity of quantifier-free formulas in Presburger arithmetic extended by propositional (Boolean) variables and function symbols with equality. This domain is a decidable one; valid formulas within it are certified as VALID by the decision box, while invalid formulas are disproved by finding and exhibiting a counterexample. Extension of the system to handle wider domains is now under consideration, but will probably result in loss of full decidability. However, even a semidecision algorithm would be of considerable utility, though this means that the system would forfeit its ability always to discover counterexamples for invalid formulas. It should be emphasized that when the user invokes special deduction techniques or auxiliary information to arrive at a tableaux proof, the Proof Supervisor keeps a record of what additional information was used, and reports that the proof of correctness was contingent on that auxiliary information or procedures.

In the RPE/1 JOVIAL Verification System only a subset of JOVIAL constructs were accomodated. These comprised assignment to simple variables and to arrays: compound, conditional and alternative statements: an iterative statement (this last following J73 conventions): and procedure calls. This RPE/1 subset was therefore a structured subset of JOVIAL. In the current RPE/2 phase we plan to accomodate a much wider subset of JOVIAL (J3-JOCIT). All but certain machine-dependent constructs are handled by the front end parser-transducer. The rest of the system will

accomodate all data- and procedure declarations, file- and table I/O, the 'exchange', 'alternative', 'switch' and 'goto' statements (among others), in addition to the features that were handled by the RPE/1 system. We expect that the system to be delivered will be a semi-practical verification tool for skilled programmers. Further work will be required, however, to make it usable by non-specialists in program verification techniques.

The RPE/1 verification system and the partially completed RPE/2 system presently reside in files at the SRI-AI (ArpaNet) DEC-10 computer. They are implemented in INTERLISP. The RPE/1 system is being translated into Maclisp, and will be transported to the RADC-MULTICS Honeywell 6180 computer facility as part of the project work of the current phase.

APPENDIX A -- JOCIT PARSER-TRANSDUCER

A Sample Dialog with the Interactive JOCIT Parser

The JOCIT program shown immediately below contains syntax errors at the lines marked by asterisks (*). This faulty program resides in a TENEX file called TP.ERRORS. We will show below the actual text of a dialog with the interactive (error-correcting) parser/transducer developed by Boyer and Moore.

[The JOCIT program:]

```
START PROC TPROG(AA,BB =ZZ) $
    ASSERTIN BB GQ 0 $
    ASSERTOUT ZZ EQ AA*BB $
    ITEM AA I 20 S $
    ITEM BB I 20 U $
    ITEM ZZ I 20 $                          *[type designator S/U missing]
    ITEM I1 I 20 P $                        *[the P is illegal here]
    BEGIN
        ZZ = 0 $
        I1 = 0 $
        LL. ASSERT ZZ EQ AA*I1 $
        IF I1 NQ BB $
        ZZ = ZZ+AA $
        GOTO LL                             *[terminator $ is missing]
    END
TERM $
```

[End of the JOCIT program]

[The next section displays interaction with the parser/transducer carried out in Lisp. As above, comments that were typed in later are enclosed in brackets []. Except for the user responses (noted marginally as such) all other text was supplied by the system.]

```
44_J(TP.ERRORS)                             [User asks for parse of the file]
Illegal lexeme  $ (of type $) in
...
I 20 S $
    ITEM BB I 20 U $
    ITEM ZZ I 20 $!                         [Diagnostics: the ! indicates that
    ITEM I1 I 20 P $                         the preceding lexem· was illegal]
    BEGIN
        ZZ = 0
```

```
A lexeme of one of the following types was expected:
S  U                                        [More system-supplied
                                             diagnostics]
What should be done?  Replace with:  "S $"
                                            [User uses R(eplace) option: only
                                             R and the string "S $" were typed
                                             by the user. He could also have
                                             typed T(eco) to invoke a text edi-
                                             tor, or typed A to abort parsing.]

                                            [The parse continues]
Illegal lexeme  P (of type P) in
...
I 20 U $
    ITEM ZZ I 20 S $
    ITEM I1 I 20 P! $                       [The next bug is discovered by
    BEGIN                                    the parser]
        ZZ = 0 $
        I1 = 0 $

A lexeme of one of the following types was expected:
S  U
What should be done?  Replace with:  "U"
                                            [User opts for Replace option:
                                             the parse continues after fix]
Illegal lexeme  END (of type END) in
...                                         [Next error is found]
$
        ZZ = ZZ+AA $
        GOTO LL
    END!                                    [The ! indicates point of fault]
TERM $

A lexeme of one of the following types was expected:
)  $  ,  LPAR\DOLL                          [The diagnostics tell user that
                                             a $ was omitted]
What should be done?  Replace with:  "$ END"
                                            [User types R and replacement
                                             string]

[At this point the parse concludes successfully, with printout of the
transduction.  We have abridged it here, in order to save space in favor
of the indented printout via PRINTDEF which follows].

(SUBPROGRAM TPROG ((AA BB) (OUT:PARS ZZ)) (ASSERTIN (GTQ BB (
DECIMAL\INTEGER "0"))) (ASSERTOUT (EQ ZZ (TIMES AA BB))) (ITEM AA ...
```

```
45_(PRINTDEF (VALUEOF 44))                    [This lined typed by user]
(SUBPROGRAM TPROG ((AA BB)
                   (OUT:PARS ZZ))
                  (ASSERTIN (GTQ BB (DECIMAL\INTEGER "0")))
                  (ASSERTOUT (EQ ZZ (TIMES AA BB)))
                  (ITEM AA (I (DECIMAL\INTEGER "20")
                              S NIL NIL NIL)
                        NIL NIL)
                  (ITEM BB (I (DECIMAL\INTEGER "20")
                              U NIL NIL NIL)
                        NIL NIL)
                  (ITEM ZZ (I (DECIMAL\INTEGER "20")
                              S NIL NIL NIL)
                        NIL NIL)
                  (ITEM I1 (I (DECIMAL\INTEGER "20")
                              U NIL NIL NIL)
                        NIL NIL)
                  (BEGIN (:= ZZ (DECIMAL\INTEGER "0"))
                         (:= I1 (DECIMAL\INTEGER "0"))
                         (LABEL LL)
                         (ASSERT (EQ ZZ (TIMES AA I1)))
                         (IF (NEQ I1 BB)
                             (:= ZZ (PLUS ZZ AA)))
                         (GOTO LL NIL)))NIL          [The final "NIL" here is
                                                      an artifact of PRINTDEF
                                                      and is not part of the
                                                      transduction]
```

[Next the user writes the transduction out to a new TENEX file by using the INTERLISP function WRITEFILE[expr; filename].

```
46_(WRITEFILE (LIST (VALUEOF 44)) 'TP.CORRECT/PARSE]
<ELSPAS>TP.CORRECT/PARSE;1

                        [END OF DIALOG]
```

[We display that whole file below]:

```
(PRIN1 (QUOTE "
WRITEFILE OF <ELSPAS>TP.CORRECT/PARSE;1 MADE BY ELSPAS ON
27-JUL-76 16:25:15
")T)
```

APPENDIX A -- JOCIT PARSER-TRANSDUCER

```
(SUBPROGRAM TPROG ((AA BB)
                   (OUT:PARS ZZ))
                  (ASSERTIN (GTQ BB (DECIMAL\INTEGER "0")))
                  (ASSERTOUT (EQ ZZ (TIMES AA BB)))
                  (ITEM AA (I (DECIMAL\INTEGER "20")
                              S NIL NIL NIL)
                        NIL NIL)
                  (ITEM BB (I (DECIMAL\INTEGER "20")
                              U NIL NIL NIL)
                        NIL NIL)
                  (ITEM ZZ (I (DECIMAL\INTEGER "20")
                              S NIL NIL NIL)
                        NIL NIL)
                  (ITEM I1 (I (DECIMAL\INTEGER "20")
                              U NIL NIL NIL)
                        NIL NIL)
                  (BEGIN (:= ZZ (DECIMAL\INTEGER "0"))
                         (:= I1 (DECIMAL\INTEGER "0"))
                         (LABEL LL)
                         (ASSERT (EQ ZZ (TIMES AA I1)))
                         (IF (NEQ I1 BB)
                             (:= ZZ (PLUS ZZ AA)))
                         (GOTO LL NIL)))
STOP                                          [STOP is the EOF marker]
```

Now, as the reader may have noticed, the above program contains serious semantic errors in addition to the syntax errors already repaired. Of course, the parser/transducer is incapable of discovering semantic errors since these are connected with the programmers intentions in writing the program. These intentions are, however, embodied in the input-, output-, and inductive assertions embedded in the program. In particular, the relation ZZ = AA*I1 is supposed to be maintained as control passes around the loop. That is what is meant by calling that relation a 'loop invariant'. Since ZZ is altered while I1 and AA are unchanged around the loop, the above relation is not maintained invariant. It happens that this inductive assertion is true on first entry to the labelled point LL, but the relation woould be destroyed after the assignment ZZ := ZZ+AA, unless AA happened to be 0.

There is another semantic error. The program will never terminate since control always passes back to the label LL, regardless of whether I1 EQ BB or not. The programmer intended, of course, to use I1 NQ BB as as a test for loop exit, as well as for determining whether I1 and ZZ are to be incremented. The control structure of the IF statement and the GOTO is quite incorrect with respect to the programmer's intentions.

In the next Appendix we show how these semantic errors can be revealed by subsequent stages in the verification process, in particular by generating verification conditions for the (transduced) program.

APPENDIX B -- DETECTION OF SEMANTIC ERRORS BY VCG

Detection and Correction of *Semantic Errors Through* Verification Conditions

Here we illustrate how the generation and inspection of verification conditions (VCs) aids in semantic debugging of a JOCIT program. It should be emphasized that the system we are using here is in a rather incomplete state. As a result the detailed form of the dialog illustrated is quite different --in particular regarding user-interaction features-- from either the existing RPE/1 verifier or the RPE/2 verifier when completed. The RPE/1 system contained an interactive deductive system (not used here) that is capable of proving the mathematical correctness of the appropriate VCs. [A mechanical proof of correctness of VCs in a slightly different form that was carried out by the RPE/1 deductive system is shown in Appendix D]. Here we content ourselves with human inspection of the VCs to discover that they are invalid, then we repair the program and regenerate VCs that are provable manually.

We redisplay the transduced program (containing semantic errors as mentioned earlier):

```
36_PP TPROG
  (SUBPROGRAM TPROG ((AA BB)
                     (OUT:PARS ZZ))
              (ASSERTIN (GTQ BB (DECIMAL\INTEGER "0")))
              (ASSERTOUT (EQ ZZ (TIMES AA BB)))
              (ITEM AA (I (DECIMAL\INTEGER "20")
                          S NIL NIL NIL)
                    NIL NIL)
              (ITEM BB (I (DECIMAL\INTEGER "20")
                          U NIL NIL NIL)
                    NIL NIL)
              (ITEM ZZ (I (DECIMAL\INTEGER "20")
                          S NIL NIL NIL)
                    NIL NIL)
              (ITEM I1 (I (DECIMAL\INTEGER "20")
                          U NIL NIL NIL)
                    NIL NIL)
              (BEGIN (:= ZZ (DECIMAL\INTEGER "0"))
                     (:= I1 (DECIMAL\INTEGER "0"))
                     (LABEL LL)
                     (ASSERT (EQ ZZ (TIMES AA I1)))
                     (IF (NEQ I1 BB)
                         (:= ZZ (PLUS ZZ AA)))
                     (GOTO LL NIL)))
TPROG
```

[Next, the user invokes the verification condition generator on TPROG, using a function appropriate to subprograms.]

APPENDIX B -- DETECTION OF SEMANTIC ERRORS BY VCG

```
37_(VCG:SUBPROGRAM TPROG)
TPROG
```

[This function returns the subprogram's name when VCG has finished, but it also sets a new variable, TPROG#, to the list of VCs. Then the user asks for an indented display of these VCs.]

```
38_PP TPROG#
  [[IMPLIES (AND (GTQ BB (DECIMAL\INTEGER "0"))
                 (ITEM AA (I (DECIMAL\INTEGER "20")
                             S NIL NIL NIL)
                       NIL NIL)
                 (ITEM BB (I (DECIMAL\INTEGER "20")
                             U NIL NIL NIL)
                       NIL NIL)
                 (ITEM ZZ (I (DECIMAL\INTEGER "20")
                             S NIL NIL NIL)
                       NIL NIL)
                 (ITEM I1 (I (DECIMAL\INTEGER "20")
                             U NIL NIL NIL)
                       NIL NIL))
            (EQ (DECIMAL\INTEGER "0")
                (TIMES AA (DECIMAL\INTEGER "0"]
   (IMPLIES (EQ ZZ (TIMES AA I1))
            (IF (NEQ I1 BB)
                (EQ (PLUS ZZ AA)
                    (TIMES AA I1))
                (EQ ZZ (TIMES AA I1]
TPROG#
```

[There are two VCs, the first expressing the validity of the inductive assertion when it is first encountered--essentially that 0 = AA*0 -- given the various hypotheses. The hypotheses consist of the collected item declarations together with the entry assertion. In this case the hypotheses are irrelevant to the conclusion, though that would usually not be the case. We anticipate that in future versions of the verifier the item declarations will be processed by the VCG into a more mathematical assertion language. The second VC expresses (a) the invariance of the inductive relation around the loop and (b) the implication relating that inductive assertion to the exit assertion on loop exit. The former (a) is mathematically false, while (b), the latter--although true--is misleading since the loop cannot exit. This is a fundamental deficiency of partial correctness proofs. We would have to supply a more complex inductive assertion here to (attempt to) prove termination, and then discover the termination error. This could be done here by using I1 as a loop counter, and adding a clause (LTQ I1 BB) to the inductive assertion. The proof that this clause is invariant would effectively bound the number of loop executions above by BB, thus demonstrating termination. Note however that the failure to increment I1 would first have to be repaired before the looping error could be detected].

[The user next edits the transduced program. This is not a recommended procedure: he should edit the original JOCIT program and then reparse it. The shortcut was used here to save space.]

APPENDIX B -- DETECTION OF SEMANTIC ERRORS BY VCG

```
39_EDIT(TPROG)                          [User invokes Lisp editor on TPROG]
edit
25*F IF PP                              [Search for the IF statement]
(IF (NEQ I1 BB)
    (:= ZZ (PLUS ZZ AA)))
26*3 MBD BEGIN]                         [Embeds the assignment in a BEGIN,]
PP
(BEGIN (:= ZZ (PLUS ZZ AA)))
27*(N (:= I1 (PLUS I1 (DECIMAL\INTEGER "1")))(GOTO LL NIL))
                                        [and appends the missing statements]
0 PP                                    [Here user asks to see the edited
(IF (NEQ I1 BB)                          IF clause]
    (BEGIN (:= ZZ (PLUS ZZ AA))
           (:= I1 (PLUS I1 (DECIMAL\INTEGER "1")))
           (GOTO LL NIL)))
28*0 PP
(BEGIN (:= ZZ (DECIMAL\INTEGER "0"))    [User instructs editor to go up one
       (:= I1 (DECIMAL\INTEGER "0"))     level and print out the BEGIN form]
       (LABEL LL)
       (ASSERT (EQ ZZ (TIMES AA I1)))
       (IF (NEQ I1 BB)
           (BEGIN (:= ZZ (PLUS ZZ AA))
                  (:= I1 (PLUS I1 (DECIMAL\INTEGER "1")))
                  (GOTO LL NIL)))
       (GOTO LL NIL))
29*(7)                                  [The extraneous GOTO is deleted]
P
(BEGIN (:= ZZ &) (:= I1 &) (LABEL LL) (ASSERT &) (IF & &))
30*OK
TPROG

40_REDO VCG:SUBPROGRAM          [The operation VCG is redone on the
TPROG                            corrected transduced program]

41_PP TPROG#                    [User asks to see the all the VCs]

  [[IMPLIES (AND (GTQ BB (DECIMAL\INTEGER "0"))
                 (ITEM AA (I (DECIMAL\INTEGER "20")
                             S NIL NIL NIL)
                       NIL NIL)
                 (ITEM BB (I (DECIMAL\INTEGER "20")
                             U NIL NIL NIL)
                       NIL NIL)
                 (ITEM ZZ (I (DECIMAL\INTEGER "20")
                             S NIL NIL NIL)
                       NIL NIL)
                 (ITEM I1 (I (DECIMAL\INTEGER "20")
                             U NIL NIL NIL)
                       NIL NIL))
            (EQ (DECIMAL\INTEGER "0")
                (TIMES AA (DECIMAL\INTEGER "0"]
```

APPENDIX B -- DETECTION OF SEMANTIC ERRORS BY VCG

```
   (IMPLIES (EQ ZZ (TIMES AA I1))
            (IF (NEQ I1 BB)
                (EQ (PLUS ZZ AA)
                    (TIMES AA (PLUS I1 (DECIMAL\INTEGER "1"))))
                (EQ ZZ (TIMES AA BB]
TPROG#
```

This concludes the dialog. The VCs can now be seen by inspection to be correct. Except for the appearance of transduced numeric constant expressions like (DECIMAL\INTEGER "0"), the RPE/1 deductive system could easily handle the proof of these VCs. The RPE/2 system, when it has been completed, will be able to deal with such expressions, treating them in accordance with JOVIAL semantics for the appropriate data types, integer, floating-point, and fixed-point numbers.

APPENDIX C -- HYPOTHETICAL DIALOG

A HYPOTHETICAL VERIFICATION SCENARIO

We present here an imaginary dialog between a user and a semiautomatic verification system. This represents a possible form that an interactive program-writing, debugging, and verification system might take. Many of the imagined features already exist (some in more primitive forms), others exist only on paper, and still others are some way off from realization.

For clarity in presentation we display user type-ins in upper-case characters, and most system responses in lower case (or mixed case). Text enclosed in square brackets has been inserted to help the reader understand what is supposed to be happening. The system uses _, +, and * as prompt characters. The user starts things off by calling the program verifier from the system executive (prompt character @).

```
[Beginning of dialog]

@VERIFIER

Verifier ready. Type ? for options.
_?

Options:
        Parser: type P
        Verification Condition Generator: type VCG
        Deductive System: type DS

_P
<loading JOCIT.PARSER>

Type F if you wish to parse an existing JOCIT file, or type I(nput) if
you wish to write a new program.

_F                          [the user opts for an existing file]

Type name of your JOCIT program file:  ARRAYSCANNER

<loading USER.ARRAYSCANNER>

<parsing ARRAYSCANNER>

Illegal lexeme found: END
at:

   BEGIN XX=AA+BB $ ZZ=AA-BB END!        [This upper-case text is printed
   TERM $                                by the system to provide diagnos-
                                         tic help]
```

APPENDIX C -- HYPOTHETICAL DIALOG

```
Parser was expecting  one of: $ * + - (
What should be done? (type ? for options)
_?
Options:
        R(eplace)
        T(eco) for TECO text editor
        TV for Display-oriented editor
        A(bort)

_R

Type string to replace END (in double quotes):   "$ END"

Parse successfully completed.  Type filename if you wish to save this
parse, else type <CR>.

_ARRAYSCAN.PARSE                          [User asks for the parsed and
                                           transduced program to be saved
                                           under the name he types in]

File has been made.  Type ? for options.

_?

Options:
        Type V for VCG
        Type P for a new parse
        Type Q(uit) to exit

_V
<loading VCG>

VCG is ready. Type ? for options.         [The new ready symbol + is
                                           indicative of the VCG]
+?
Type L(ast) if you want to generate VCs for last parse:
type F(ile) if you want to VCG another existing file.

+L

<generating VCs for ARRAYSCAN.PARSE>

VCs completed. Type filename or <CR>:     [the user types a carriage return
                                           to indicate he doesn't wish to
                                           save these VCs now.]

Do you wish to invoke Deductive System? If so, type DS: else ? for other
options.
+DS

<loading Deductive.System>

Ready.  Type L(ast) for last-generated VCs, or ? for other options.
```

APPENDIX C -- HYPOTHETICAL DIALOG

```
*L                                          [* is the ready symbol of the DS]

<Trying to prove VCs for ARRAYSCAN.PARSE>

Unable to prove VC#3 (for-loop: transduced code starts:

        (FOR I (0 2 100) (ASSERT T) ...)

Do you wish to modify the inductive assertion?

*Y

Options:
        Type E(dit) to invoke LISP editor
        Type A(bort) to abort proof of this clause, and continue proof
        Type Q(uit) to proof entirely

*E

Edit

**pp

(FOR I (0 2 100)
       (ASSERT T)
       (BEGIN (:= ZZ (PLUS ZZ I))
              (IF (GT ZZ MAXZZ)
                  (:= ZZ MAXZZ))))

**R T (AND (LTQ ZZ MAXZZ)(LTQ 0 I)(LTQ I 100)))

**p

(FOR I (0 2 100) (ASSERT (AND & & &)) (BEGIN & &)))

**OK

<Continuing proof of VCs for ARRAYSCAN>

Found embedded subprocedure "PROCESSELEMENTS" inside ARRAYSCAN.
Do you wish to verify subprocedure now? (type N for Now) or
(type D for Defer).

*D

Verification of subprocedure PROCESSLELEMENTS has been deferred.

<Continuing proof of VCs for ARRAYSCAN>

Proof of VCs for ARRAYSCAN completed
Subject to correctness of assertions for PROCESSELEMENTS.

Do you wish to verify PROCESSELEMENTS now?
```

APPENDIX C -- HYPOTHETICAL DIALOG

```
*Y

<Generating VCs for PROCESSELEMENTS>

VCG of PROCESSELEMENTS completed.

<Verifying VCs for PROCESSELEMENTS>

Proof of VCs for ARRAYSCAN completed
No outside assumptions used in proof.

Do you wish to save the proof trace?

*Y

Type name of file:  (<CR> defaults to progname.PROOFTRACE)

                                        [User types carriage return]
Saved on PROCESSELEMENTS.PROOFTRACE

Other verifications?

*N

<VERIFIER exited)

@
```

APPENDIX D--SAMPLE GOAL-DRIVEN PROOF OF VCS

PROOF OF VCS BY THE RPE/1 GOAL-DRIVEN DEDUCTIVE SYSTEM

This shows a proof trace for validity of the three VCs exhibited directly below, as the value of TPROG#. The deductive sysem used here is a goal-driven system that was part of the RPE/1 verifier.

```
184_PP TPROG#
  [(IMPLIES (GTQ BB 0)
            (EQ 0 (TIMES AA 0)))
   [IMPLIES (EQ ZZ (TIMES AA I1))
            (IMPLIES (NEQ I1 BB)
                     (EQ (PLUS ZZ AA)
                         (TIMES AA (PLUS I1 1]
   (IMPLIES (EQ ZZ (TIMES AA I1))
            (IMPLIES (NOT (NEQ I1 BB))
                     (EQ ZZ (TIMES AA BB]
TPROG#

185_(FOR CLAUSES IN TPROG# (PROVE CLAUSES))   [User calls for proof of
Trying-to-prove:                               the 3 VCs]

  (IMPLIES (GTQ BB 0)
           (EQ 0 (TIMES AA 0)))                [The first VC]

Equality-true-by-simplification

IMPLIES-proved-TRUE-by-simplification
Proof-is-successful!

Trying-to-prove:

  [IMPLIES (EQ ZZ (TIMES AA I1))               [The second VC]
           (IMPLIES (NEQ I1 BB)
                    (EQ (PLUS ZZ AA)
                        (TIMES AA (PLUS I1 1]
To-prove:
  [IMPLIES (EQ ZZ (TIMES AA I1))
           (OR (EQ BB I1)
               (EQ (PLUS AA ZZ)
                   (TIMES AA (PLUS 1 I1]
we-try-to-assert:

(EQ ZZ (TIMES AA I1))

and-then-try-to-prove:

  [OR (EQ BB I1)
      (EQ (PLUS AA ZZ)
          (TIMES AA (PLUS 1 I1]
(Asserting: (EQ ZZ (TIMES AA I1)))
CONSISTENCY-CHECK-WAS-BYPASSED
Consistent
Trying-to-prove:
```

```
  [OR (EQ BB I1)
      (EQ (PLUS AA ZZ)
          (TIMES AA (PLUS 1 I1]
To-prove:
  [IMPLIES (NEQ BB I1)
           (EQ (PLUS AA ZZ)
               (TIMES AA (PLUS 1 I1]
we-try-to-assert:

(NEQ BB I1)

and-then-try-to-prove:

  (EQ (PLUS AA ZZ)
      (TIMES AA (PLUS 1 I1)))
(Asserting: (NEQ BB I1))
CONSISTENCY-CHECK-WAS-BYPASSED
Consistent
Trying-to-prove:

  (EQ (PLUS AA ZZ)
      (TIMES AA (PLUS 1 I1)))
Checking-if:
  (EQ (PLUS AA ZZ)
      (TIMES AA (PLUS 1 I1)))
was-asserted-at-the-context:
(11 10)
or-at-some-parent-context
It-was-not-previously-asserted

Invoking-fast-transitivity-routine
(Want to simplify: (EQ (PLUS AA ZZ) (TIMES AA (PLUS 1 I1))) ?)
YES
ok
Applying-SIMPLIFY-to:

  (EQ (PLUS AA ZZ)
      (TIMES AA (PLUS 1 I1)))
Simplified-to:

  (EQ (PLUS AA ZZ)
      (PLUS AA (TIMES AA I1)))                    [The simplifier uses
Checking-if:                                       the distributive law]
  (EQ (PLUS AA ZZ)
      (PLUS AA (TIMES AA I1)))
was-asserted-at-the-context:
(11 10)
or-at-some-parent-context
It-was-not-previously-asserted
(Trying-to-verify-an-EQUALITY-relation (EQ (PLUS AA ZZ) (PLUS AA (TIMES
AA I1))))
(Trying-to-prove-relation: (EQ (PLUS AA ZZ) (PLUS AA (TIMES AA I1)))
Want-to-subst-for-some-subexpr-of: (PLUS AA ZZ) ?)
YES
ok                                                 [User elects to invoke
                                                    a substitution]
```

APPENDIX D--SAMPLE GOAL-DRIVEN PROOF OF VCS

```
(type-in::SUBST:L<subexpr-to-be-replaced> broken)
186:SUBST:L(ZZ)
(1 POSSIBLE-SUBSTITUTIONS)
(substitute (TIMES AA I1) for ZZ in ((PLUS AA ZZ)) ?)
YES
ok
(PLUS AA (TIMES AA I1))
187:GO
type-in::SUBST:L<subexpr-to-be-replaced> = finished
(For-some-subexpr-of: (PLUS AA (TIMES AA I1)) ?)
NO
refused-by-user

Equality-true-by-simplification
Proof-is-successful!

Proof-is-successful!

Proof-is-successful!                          [This completes proof of VC2]

Trying-to-prove:

  [IMPLIES (EQ ZZ (TIMES AA I1))              [The third VC]
           (IMPLIES (NOT (NEQ I1 BB))
                    (EQ ZZ (TIMES AA BB]
To-prove:
  [IMPLIES (EQ ZZ (TIMES AA I1))
           (OR (NEQ BB I1)
               (EQ ZZ (TIMES AA BB]
we-try-to-assert:

(EQ ZZ (TIMES AA I1))

and-then-try-to-prove:

  (OR (NEQ BB I1)
      (EQ ZZ (TIMES AA BB)))
(Asserting: (EQ ZZ (TIMES AA I1)))
CONSISTENCY-CHECK-WAS-BYPASSED
Consistent
Trying-to-prove:

  (OR (NEQ BB I1)
      (EQ ZZ (TIMES AA BB)))
Checking-if:
  (OR (NEQ BB I1)
      (EQ ZZ (TIMES AA BB)))
was-asserted-at-the-context:
(12)
or-at-some-parent-context
It-was-not-previously-asserted
To-prove:
  (IMPLIES (EQ BB I1)
           (EQ ZZ (TIMES AA BB)))
we-try-to-assert:
```

```
(EQ BB I1)

and-then-try-to-prove:

  (EQ ZZ (TIMES AA BB))
(Asserting: (EQ BB I1))
CONSISTENCY-CHECK-WAS-BYPASSED
Consistent
Trying-to-prove:

  (EQ ZZ (TIMES AA BB))
Checking-if:
  (EQ ZZ (TIMES AA BB))
was-asserted-at-the-context:
(13 12)
or-at-some-parent-context
It-was-not-previously-asserted

Invoking-fast-transitivity-routine
(Want to simplify: (EQ ZZ (TIMES AA BB)) ?)
NO
refused-by-user

(Trying-to-verify-an-EQUALITY-relation (EQ ZZ (TIMES AA BB)))
(Trying-to-prove-relation: (EQ ZZ (TIMES AA BB))
Want-to-subst-for-some-subexpr-of: ZZ ?)
NO                                          [The user could have said YES,
refused-by-user                              and asked for a substn on ZZ]
(For-some-subexpr-of: (TIMES AA BB) ?)
YES
ok

(type-in::SUBST:R<subexpr> broken)
188:SUBST:R(BB)
(1 POSSIBLE-SUBSTITUTIONS)
(substitute I1 for BB in ((TIMES AA BB)) ?)
YES
ok
(TIMES AA I1)
189:GO
type-in::SUBST:R<subexpr> = finished
Proof-is-successful!

Proof-is-successful!

Proof-is-successful!

        [This completes the machine proof of the three VCs]
```

System Development Corporation

A PROGRAM VERIFIER WITH ASSERTIONS IN TERMS OF ABSTRACT DATA

VAL SCHORRE

SP-3841

ABSTRACT

This paper describes a program verifier whose basic unit of verification is a module consisting of data and procedures. The module hides the data by allowing the data to be changed and accessed only by calling the procedures. In specifying a module to the verifier, the data is thought of abstractly in terms of information content rather than representation for efficient storage and retrieval. The abstract data is represented by a set of state variables. A notation based on pure LISP is used to write the entry and exit assertions for the procedures of the module. The state variables can contain functions as well as other forms of data. The language of the verifier does not provide for explicitly defining the type of the data in the state variables. Instead, the type of the data is determined by the use of these variables in the entry and exit assertions. The actual representation of data is described by declaration in the procedural language used for implementation. The notation based on pure LISP is further used to specify the mapping from the abstract data into the actual data.

The method used by the verifier is based on Floyd's ideas of program semantics combined with the approach which Boyer and Moore used to prove theorems about LISP programs.

The verifier described is currently being implemented at System Development Corporation in our proprietary system CWIC (Compiler for Writing and Implementing Compilers). It is running and has verified some procedures, but it does not have all the features described.

The paper first describes the verifier from the standpoint of a user by giving a line by line description of a sample module. It then presents an overall discussion of the implementation.

1. INTRODUCTION

The concept of a module, as used in this paper, is that of a programming unit which consists of data and procedures. It hides the data by allowing it to be changed and accessed only by calling its procedures. This is similar to the concept of a module, as expressed by Parnas[19], and to the concept of a cluster, as described by Liskov and Zilles[15].

The basic ideas which are used in verification today came from a paper on the semantics of programming languages by Floyd[6]. The current trend of applying these ideas began with a software tool called a program verifier by King[14]. This was soon followed by a number of other verifiers which are described in the following papers: Good[9], Elspas, Green, Levitt, Waldinger[5], Deutsch[4], Schorre[22], Good, London, Bledsoe[11], and Henke, Luckham[12].

In working out examples for such verifiers, the author has found it awkward to express exit assertions in terms of the arrays used by the procedures. In describing a dictionary module, it is natural to say that the procedure PUT puts information into the dictionary and that the procecure GET gets it out. It is not relevant whether the dictionary is hash coded or sorted in ascending order for a binary search. Nevertheless, in describing the PUT routine to any of the aforementioned verifiers it is necessary to describe its effect on the actual arrays used in the implemention. The result is that the retrieval algorithm used by the dictionary module permeates the exit assertion for PUT.

A similar desire for abstract specification has been expressed by Good[10]. He has shown examples of procedures with abstract specification, but not in a form for automatic processing by a verifier.

An abstract specification of the dictionary module will now be given, followed by an implementation that uses the linear search algorithm. This specification and implementation are intended as input to the verifier being implemented at System Development Corporation (SDC), although some changes may be required before they are actually processed. Implementation using other algorithms are possible, but will not be given in this paper. The verifier is explained by detailed consideration of the dictionary module example. The input to the verifier is given in Figures 1 through 3. Figure 1 is the specification of the module in terms of abstract data. Figure 2 showns the declarations for the concrete data used in the linear search implementation, and a mapping from the abstract data onto the concrete data. Figure 3 shows the routines of this implementation written in the procedural language used by the verifier. These three figures are explained in Section 2 following.

```
.MODULE
.SPECIFICATION
.STATE DICTIONARY, STACK;
.INITIAL DICTIONARY = .LAMBDA X, Y (0) & STACK = .NIL;
.PROCEDURE PUSH ();
    .ENTRY .TRUE:
    .EXIT DICTIONARY'=DICTIONARY & STACK'=DICTIONARY " STACK;
.PROCEDURE POP ():
    .ENTRY STACK ≠ .NIL;
    .EXIT DICTIONARY'=STACK.1 & STACK'=STACK:2;
.PROCEDURE PUT(F, X, Y):
    .ENTRY .TRUE;
    .EXIT STACK' = STACK &
    DICTIONARY' = .LAMBDA I, J(
                        .IF I=F & J=X
                             .THEN Y
                             .ELSE DICTIONARY (I, J)  .END);
.PROCEDURE GET(F, X; Y):
    .ENTRY .TRUE;
    .EXIT STACK'=STACK & DICTIONARY'=DICTIONARY &
    Y'=DICTIONARY(F, X);
```

V. Schorre
Figure 1.

```
.IMPLEMENTATION
.DECLARE DICT_F[1000], DICT_X[1000], DICT_Y[1000], STACKX[100],
    RUN, LEVEL;
.DEFINE
SEARCH(F, X, N) =
    .IF N < 0
        .THEN 0
    .OR IF F=DICT_F[N] & X=DICT_X[N]
        .THEN DICT_Y[N]
    .OR IF .TRUE
        .THEN SEARCH(F, X, N-1) .END;
TAIL(W) =
    .IF W=0
        .THEN .NIL
        .ELSE .LAMBDA F, X(
                SEARCH(F, X, STACKX[W]-1)) "
                TAIL(W-1) .END);
.MAP
DICTIONARY = .LAMBDA F, X(SEARCH(F, X, RUN-1));
STACK = TAIL(LEVEL);
```

V. Schorre
Figure 2.

```
.ASSERT 0≤RUN<1000 & 0≤LEVEL<100 &
    ∀ I(0≤I≤LEVEL ⊃ 0≤STACK[I]<1000);
.INITIAL
    RUN := 0;  LEVEL := 0;  STACKX[0] := 0;
    .RETURN
.PROCEDURE PUSH:
    .ERROR UNLESS LEVEL < 99;
    LEVEL := LEVEL + 1;  STACKX[LEVEL] := RUN;
    .RETURN
.PROCEDURE POP:
    RUN := STACKX[LEVEL]; LEVEL := LEVEL - 1;
    .RETURN
.PROCEDURE PUT:
    .DECLARE H;
    .ERROR UNLESS RUN < 999;
    H := STACKX[LEVEL];
    .LOOP ∀ I(STACKX[LEVEL]≤I<H' ⊃ F≠DICT_F[I] ∨ X≠DICT_X[I]) &
    STACKX[LEVEL]≤H'
   .WHILE H<RUN+1 & (F≠DICT_F[H] & X≠DICT_X[H]) .DO
         H := H + 1; .END
    .IF H = RUN+1
         .THEN DICT_Y[H] := Y;
         .ELSE DICT_F[RUN] := F;  DICT_X[RUN] := X;
    DICT_Y[RUN] := Y:  RUN := RUN+1;  .END
    .RETURN
.PROCEDURE GET:
    .DECLARE H;
    H := RUN - 1;
    .LOOP ∀ I(H'<I≤RUN-1 ⊃ F≠DICT_F[I] ∨ X≠DICT_X[I]) &
    H'<RUN-1
    .WHILE 0<H & (F≠DICT_F[H] ∨ X≠DICT_X[H]) .DO
         H := H-1; .END
    .IF F=DICT_F[H] & X=DICT_X[H]
         .THEN Y := DICT_Y[H];
         .ELSE Y := 0; .END
    .RETURN
.FINISH
```

V. Schorre

Figure 3.

2. EXAMPLE OF THE DICTIONARY MODULE

2.1 INFORMAL DESCRIPTION OF THE PROBLEM

The dictionary of the example used in this paper is like a function whose values are specified one at a time by the user. When the user wishes to assign the value Y to the function F(X), he calls the procedure PUT(F, X, Y). When he wishes to get the value of F(X), which he has previously assigned by using the PUT procedure, and store it in Y, he calls the procedures GET(F, X; Y). There are two additional procedures, PUSH and POP. PUSH remembers the current state of the dictionary, but does not alter it in any way; POP restores the state of the dictionary to the state in which it was at the corresponding PUSH. If more than one value has been assigned to F(X), GET returns the last such value that has not been removed by POP. If no value is currently entered for F(X), GET returns zero.

2.2 SPECIFICATION OF THE DICTIONARY MODULE IN TERMS OF ABSTRACT DATA

Figure 1 gives the dictionary module specification. No attempt is made to give a complete definition of the language in which this example is written; rather, this example specification will be discussed line by line.

```
.MODULE
.SPECIFICATION
```

The key word .MODULE signals the verifier to begin processing a new module. This key word is always followed by the key word .SPECIFICATION, because the first information that the verifier needs about a module is its specification.

.STATE DICTIONARY, STACK;

This module has two state variables, DICTIONARY and STACK. The variable DICTIONARY contains a function with two arguments. Because DICTIONARY is a state variable, it may contain a different function after execution of a procedure than it did before execution. Consider the procedure call PUT(F, X, Y). After its execution the value of the function in the variable DICTIONARY for the arguments F and X is Y. This can be expressed by the notation DICTIONARY(F, X) = Y. F may be thought of as an attribute, X as an object with that attribute, and Y as the value of that attribute. The variable STACK always contains a list of functions, each having two arguments.

Type declarations, like those in the language PASCAL described by Wirth[27], would be a useful feature to add to the verifier. Using them, the information given in the preceeding paragraph could be expressed formally to the verifier. This feature has not been included in the present version of the verifier because it is not needed to accomplish the objective of making the specification independent of the implementation.

.INITIAL DICTIONARY = .LAMBDA X, Y(0) & STACK = .NIL;

Each module contains an initialization procedure which will be executed after the entire program has been loaded and before the main program is executed. The purpose of this procedure is to initialize the data protected by the module. The expression above will be true immediately after the initial procedure for the module has been executed. The value of the state variable DICTIONARY is a function of two arguments which always returns 0. The value of the state variable STACK is the empty list, which in the language of the verifier is denoted by .NIL.

```
.PROCEDURE PUSH():
    .ENTRY .TRUE;
    .EXIT DICTIONARY'=DICTIONARY & STACK'=DICTIONARY " STACK;
```

This is the specification of the procedure PUSH. The dictionary being described has a push-down stack capability, which would be useful in compiling a block-structured language such as ALGOL 60. Each procedure is described by an entry condition and an exit condition. When the verifier processes the implementation of a procedure, it checks to see that if the entry condition is true immediately before the procedure is entered, then the exit condition will be true immediately after it is exited. When processing a call to a procedure, the verifier checks to see that the entry assertion is satisfied, and then assumes the exit assertion. Thus the entry assertion represents some restriction on the arguments of the procedure, as well as restrictions on global data at the time of call. The procedure PUSH has no restriction, so its entry assertion is .TRUE. The exit assertion involves the comparison of the state variables at two points in time--before entry into the procedure and after exit from the procedure. The state variable without the prime refers to the value of that variable before entry to the procedure. The state variable with the prime refers to the value of that variable after exit from the procedure. Thus DICTIONARY' = DICTIONARY means that the value of the state variable DICTIONARY, after exit from the procedure, is the same as it was before entry. The next part of the exit assertion is as follows:

```
STACK' = DICTIONARY " STACK
```

The double quote(") is used as the infix form of the LISP function CONS. The final value of the variable STACK is the list obtained by putting the original value of the dictionary onto the front of the list which was originally in STACK.

```
.PROCEDURE POP():
    .ENTRY STACK ≠ .NIL;
    .EXIT DICTIONARY'=STACK.1 & STACK'=STACK:2;
```

Notice that the procedure POP has a restriction on entry. It would not make sense to pop a stack if it were empty to begin with. The exit assertion makes use of the post-fix operator .1, which is the LISP function CAR, in other words, the first element of the list. The post-fix operator :2 is the LISP function CDR, in other words, the remainder of a list after the first element has been removed. The affect of the procedure POP is to set the state variable DICTIONARY to the first element of the list in STACK, and to set STACK to the rest of that list.

```
.PROCEDURE PUT(F, X, Y):
    .ENTRY .TRUE;
    .EXIT STACK' = STACK &
    DICTIONARY' = .LAMBDA I, J(
                        .IF I=F & J=X
                            .THEN Y
                            .ELSE DICTIONARY (I, J)  .END);
```

The procedure PUT has no restriction on entry. On exit, the value of the state variable STACK is unchanged. The value of the state variable dictionary upon exit is specified by a lambda expression. A lambda expression is a way of denoting a function. The variables I and J are local to the lambda expression, and represent the two arguments of the function being expressed. The value returned by this function is given by a conditional expression. If the arguments of this function are the same as the first two arguments of the procedure PUT, then the value of this function will be the third argument of PUT, otherwise, the value will be the same as for the function which was in the state variable DICTIONARY on entry to PUT. This use of a lambda expression and a conditional expression will seem natural to the reader who is already familiar with LISP.

```
.PROCEDURE GET(F, X; Y):
    .ENTRY .TRUE;
    .EXIT STACK'=STACK & DICTIONARY'=DICTIONARY &
    Y'=DICTIONARY(F, X);
```

The procedure GET has two formal input parameters, F and X, and a formal input/output parameter Y. This specification does not completely meet the goal of being implementation independent. The language in which the dictionary module is to be implemented allows procedures to have two types of parameters--input parameters and input/output parameters. The input parameters are written first and separated from the input/output parameters by a semi-colon (;). If there are no input/output parameters, then the semi-colon is omitted, as in the case of the procedures already discussed. A procedure can access the value of its input parameters, but cannot change their value. A procedure can both access and change the value of its input/output parameters.

The concept of module, as used in this paper, includes the notion that the procedures in the module form its interface with the outside world. In specifying a module, its interface is also specified. This interface can be language dependent and to a lesser degree implementation dependent. A module which has been specified as containing certain procedures could not be implemented in a version of COBOL which does not recognize the concept of procedure. The procedure GET could not be implemented according to the above specifications in a language which allows only input parameters, although the specification could be easily modified to allow implementation in such a language.

According to the exit assertion, the state variables STACK and DICTIONARY are unchanged by the procedure GET. The input/output parameter Y is set to the value obtained by applying the function contained in the state variable DICTIONARY to the two input parameters F and X.

2.3 MAPPING FROM ABSTRACT DATA INTO CONCRETE

This section describes Figure 2 line by line.

.IMPLEMENTATION

This line marks the beginning of that part of the module which is truly implementation dependent.

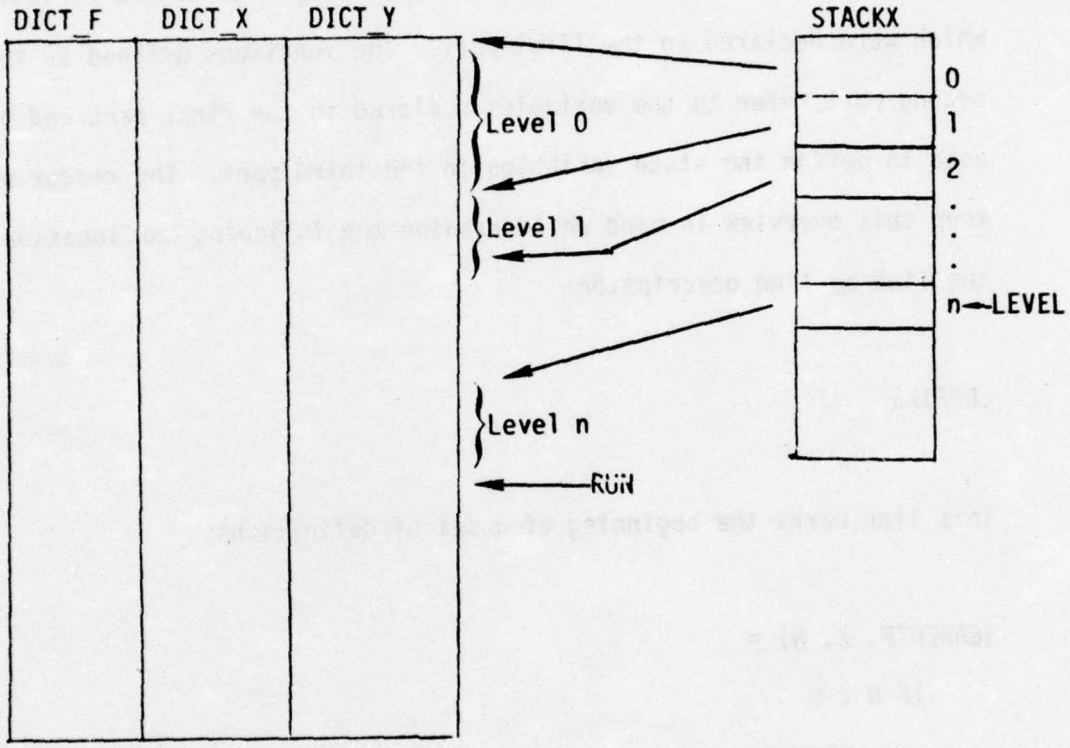
.DECLARE DICT_F[1000], DICT_X[1000], DICT_Y[1000], STACKX[100], RUN, LEVEL;

This is a declaration in the procedural language in which the module is being implemented. The only data which the language processes are integers and one-dimensional arrays of integers. Thus, RUN and LEVEL are integer

variables. DICT_F, DICT_X, and DICT_Y are one-dimensional arrays of integers, each 1000 elements long. STACKX is a one-dimensional array of 100 integers. Array indexing begins with 0 and goes to one less than the number of elements in the array. RUN and LEVEL are simple variables with integer values. Figure 4 shows how information is represented in terms of the concrete data of the module. The dictionary is represented by the parallel arrays DICT_F, DICT_X, and DICT_Y. Items are entered into these arrays one after another. RUN is a pointer to the next available location in these arrays. STACKX is a stack of values of RUN, which is used by PUSH and POP for saving and restoring the dictionary. LEVEL is a pointer to the last item used in STACKX.

Figure 4

An abstract dictionary is represented by the information in the parallel arrays above some pointer. Thus the function stored in DICTIONARY is represented by all the information in the parallel arrays above the pointer RUN. The list of functions in STACKX is represented by the information in the parallel arrays above the corresponding pointers in STACKX. Of course, STACKX is upside down with respect to STACK. The top element in STACKX points to the top elements in the parallel arrays so that there is no information above it. This pointer represents the function that always returns 0 and which is at the bottom of the list in STACK.



V. Schorre

Figure 4.

If the reader refers back to Figure 2, he will see it divided into three parts. The first part is the declarations discussed above. The second part is a group of function definitions. The third part is the definition of the state variables DICTIONARY and STACK in terms of the variables which were declared in the first part. The functions defined in the second part refer to the variables declared in the first part and are used to define the state variables in the third part. The reader should keep this overview in mind while reading the following continuation of the line by line description.

```
.DEFINE
```

This line marks the beginning of a set of definitions.

```
SEARCH(F, X, N) =
    .IF N < 0
        .THEN 0
    .OR IF F=DICT_F[N] & X=DICT_X[N]
        .THEN DICT_Y[N]
    .OR IF .TRUE
        .THEN SEARCH(F, X, N-1)   .END;
```

The function SEARCH searches the parallel arrays DICT_F and DICT_X in last-in/first-out order, beginning with the N-th entry. If it does not find a match, it returns 0. If it does find a match, it returns the corresponding entry from the array DICT_Y, otherwise, it calls itself recursively to continue the search.

```
TAIL(W) =
    .IF W=0
        .THEN .NIL
        .ELSE .LAMBDA F, X(
            SEARCH (F, X, STACKX[W]-1))  "
            TAIL(W-1)  .END;
```

TAIL(W) is the list of the bottom W dictionaries on the stack. The argument W must be a non-negative integer. When the function is called with 0, it returns the empty list .NIL. When it is called with a positive integer W, it returns a list which it constructs by putting a function expressed as a lambda expression onto the front of the list obtained by calling itself recursively with W-1. The function represents the W-th dictionary and is constructed by calling the function SEARCH to search the parallel arrays.

.MAP

The keyword .MAP marks the beginning of the definitions of the abstract state variables in terms of the concrete data.

```
DICTIONARY = .LAMBDA F, X(SEARCH(F, X, RUN-1));
```

The state variable DICTIONARY is a function expressed as a lambda expression. The recursive function SEARCH is called to search the parallel arrays for F and X beginning just above the pointer RUN.

```
STACK = TAIL(LEVEL);
```

The state variable STACK is defined by calling the recursive function TAIL, giving it the pointer to the last element in STACKX as argument.

2.4 IMPLEMENTATION OF THE DICTIONARY MODULE IN A PROCEDURAL LANGUAGE

This section describes Figure 3 in detail.

```
.ASSERT 0≤RUN<1000 & 0≤LEVEL<100 &
    ∀ I(0≤I≤LEVEL > 0<STACKX[I]≤1000);
```

This is an assertion about the concrete data of the module. It is to hold true immediately after initialization, and if it holds true just before any procedure of the module is entered, then it will hold true just after that procedure is exited. In processing the module, the verifier will prove that this assertion holds true immediately after initialization. It will assume that it holds true just before entering a procedure, and prove that it holds true just after exiting. When processing a call to one of the procedures of this module, the verifier will prove that this assertion holds before the call, and then assume that it holds afterwards.

This assertion is similar to the inductive assertions on loops which are discussed later in this paper.

```
.INITIAL
    RUN := 0;  LEVEL := 0;  STACKX[0] := 0;
    .RETURN
```

This is the code to initialize the module. It will be executed after the program is executed, but before control goes to the main program. The verifier proves that both the initial assertion and the assertion about concrete data hold true after execution of this code.

```
.PROCEDURE PUSH:
    .ERROR UNLESS LEVEL < 99;
    LEVEL := LEVEL + 1; STACKX[LEVEL] := RUN;
    .RETURN
```

This is written in a typical procedural language. The meaning of the error statement is to abort if the boolean expression LEVEL < 99 is not true. The verifier is only proving that the program meets its specification if it terminates normally. Nothing is said about the case in which the program aborts or gets into an infinite loop.

It is interesting to compare the entry assertion, which is part of the specification, to the error statement in the procedure. The error statement describes an implementation restriction on the procedure. The entry assertion is checked by the verifier, and has no code generated for it. The error expression is assumed by the verifier, because the compiler generates code to abort if it is not the case. The error statement is usually used to express a storage limitation. In this case, the limit is a maximum of 100 elements in the stack. If this module had been implemented in a language with dynamic storage allocation, it might have requested additional storage for the stack and left it up to the storage allocation routine to abort in case there was insufficient space available. It would not have been appropriate to describe in the specification of PUSH what error message would be generated in case of insufficient storage, because that would have forced an early decision about whether the stack was to be allocated as a fixed block or in some general pool.

```
.PROCEDURE POP:
    RUN := STACKX[LEVEL];  LEVEL := LEVEL-1;
    .RETURN
```

This procedure contains no new features, and needs no special explanation.

.PROCEDURE PUT:

The formal parameters, F, X, and Y, are not shown here since they have already been given in the specification of PUT.

```
.DECLARE H;
```

This is a declaration of the local variable H as a simple integer variable.

```
.ERROR UNLESS RUN < 999;
H := STACKX[LEVEL];
```

The error statement has already been discussed, and the assignment is similar to those which have been seen before.

```
.LOOP ∀ I(STACKX[LEVEL]≤I<H' ⊃ F≠DICT_F[I] ∨ X≠DICT_X[I]) &
STACKX[LEVEL]≤H'
.WHILE H<RUN+1 & (F≠DICT_F[H] & X≠DICT_X[H]) .DO
    H := H+1; .END
```

This is the first loop statement that has been encountered. It differs from the usual loop statement of a procedural language in that there is an inductive assertion between the key words .LOOP and .WHILE, as well as the usual loop termination condition between the key words .WHILE and .DO. The inductive assertion is like a bookean expression, but it can also contain universal and existential quantifiers. Unprimed variables in the inductive assertion represent their value before the loop was first entered; primed variables represent their value at some arbitrary time when control has reached the top of the loop. The verifier must prove that the inductive assertion is true when control first enters the loop, and that if it is true at the beginning of some arbitrary time through the loop, then it will be true at the end.

In a future version of the verifier, we hope to eliminate much of the need for the user to write inductive assertions. One way to do this is to have them automatically generated by the verifier, using techniques such as those described by Wegbreit[26].

Another approach is to use high level-language constructs, as was suggested by Gerhart[7], although not necessarily APL constructs. For example, the author has found that the programmer can supply most of the invariants needed to prove that subscripts are within bounds by specifying subranges in the declaration of variables. This subrange feature is currently in a number of languages, such as PASCAL, which is described by Wirth[27].

The remainder of the procedure PUT, as well as the entire procedure GET, involves nothing new, so the line by line discussion need not be continued.

3. THE VERIFIER--CURRENT STATUS

The preceeding example is to be processed by a verifier which combines the Floyd approach of inductive assertions with the techniques of proving theorems about recursive functions, as originated by Boyer and Moore[2], and extended by Moore[18]. As in the Floyd approach, procedures are described by entry and exit assertions. This is appropriate, because the objective is to handle procedures which have side effects on global data. The current verifier further follows the Floyd approach by requiring that the user supply inductive assertions. The Floyd approach involves a symbolic execution--or possibly a backward substitution--between each pair of assertions. Deutsch[4] improved this by doing a symbolic execution

beginning at each assertion. When a branch point is reached, the execution splits. Thus common initial parts of paths are handled together. SDC's verifier does a single symbolic exeuction of an entire procedure. It evaluates expressions in terms of the values which the variables had upon entry to the procedure, or--in case of variables killed in loops--in terms of the value which the variable had at the beginning of some arbitrary execution of the loop.

SDC's verifier uses a hash coding technique, similar to that of Deutsch[4], to build information as it does the symbolic execution. Deutsch must clear his dictionary before he starts a symbolic exeuction at a new assertion. When SDC's verifier reaches the inductive assertion at the beginning of the loop, it generates new values for the variables killed in the loop. This is a technique taken from global program optimization as described by Cocke and Schwartz[3] pages 252-283. No information need be removed from the dictionary at this point. Information about the variables killed in the loop becomes irrelevant due to the new value of these variables. Information not involving killed variables remains in the dictionary and does not have to be re-entered, as with Deutsch's verifier.

SDC's verifier accepts only structured programs, that is, programs without GO TO statements. The reader will recall that SDC's method of symbolic execution does require a knowledge of loop structure because it assigns new values to variables killed in loops. If one wished to encourage bad programming practice, one could program the verifier to discover loops by interval analysis. This type of analysis has been described by Sites[23] page 7-16 in connecton with program verification. It is described in more detail in relation to global program optimization by Cocke and Schwartz[3]

pages 284-303 and in still greater detail by Schaefer[21] chapters 3-7. This latter book gives a very rigorous mathematical treatment.

4. CONCLUSIONS

The author believes that a verifier is an important tool to be used where exceptionally high reliability is demanded. Programmers using this tool will need enough background in mathematics or symbolic logic to understand what a proof is.

There has been a great deal of progress over the past few years in the status of program verification software. This paper has described one of these tool as it is currently being implemented at SDC.

REFERENCES AND BIBLIOGRAPHY

[1] Bledsoe, W. W. and Good, D. I. and London, R. L. 1974. An interactive program verification system, U.S.C. Information Science Institute Report ISI/RR-74-22, APRA Order No. 2223, October 1974.

[2] Boyer, Robert S. and Moore, J Strother 1975. Proving theorems about LISP functions, Journal ACM, Vol. 22, No. 1, January 1975, pp. 129-144.

[3] Cocke, John and Schwartz, J. T., 1969. Programming languages and their compilers, Courant Institute of Mathematical Sciences, New York University.

[4] Deutsch, L. P., 1973. An interactive program verification system, Ph.D. thesis, University of California at Berkeley. Also Xerox Report No. CSL-73-1, May 1973.

[5] Elspas, B., Green, M. W., Levitt, K. N., and Waldinger, R. J., 1972. Research on interactive program-proving techniques, Stanford Research Institute, May 1972.

[6] Floyd, R. W., 1967. Assigning meanings to programs, in Procedings of a Symposium in Applied Mathematics, Vol. 19--Mathematical Aspects of Computer Science, Schwartz, J. T. (ed.), pp. 19-32.

[7] Gerhart, S., 1972. Verification of APL programs, Ph.D. thesis, Carnegie-Mellon University.

[8] German, Steven M. and Wegbreit, Ben 1975. A synthesizer of inductive assertions, Xerox Palo Alto Research Center, 3180 Porter Drive, Palo Alto, Calif. 94304, January 1975.

[9] Good, D. I., 1970. Toward a man-machine system for proving program correctness, Ph.D thesis, University of Wisconsin, 1970. Also Computation Center Report TSN-11, University of Texas at Austin, June 1970.

[10] Good, D. I., 1975. Provable programming, International Conference on Reliable Software, Los Angeles, Calif. April 1975, pp. 411-419.

[11] Good, D. I. and London, R. L., and Bledsoe, W. W. 1975. An Interactive Program Verification System, International Conference on Reliable Software, Los Angeles, Calif. April 1975, pp. 482-492.

[12] Henke, F. W. von and Luckham, D. C. 1975. A Methodology for Verifying Programs. International Conference on Reliable Software, Los Angeles, Calif. April 1975, pp. 156-164.

[13] HOARE, C. A. R., 1969. An axiomatic basis for computer programming, Comm. ACM, Vol. 12, No. 10, October 1969, pp. 576-580, 583.

[14] King, J. C., 1970. A program verifier, Ph.D. thesis, Carnegie-Mellon University.

[15] Liskov, Barbara, and Zilles, Stephen, 1974. Programming with abstract data types, ACM SIGPLAN Symposium on Very High Level Languages, March 28-29, Santa Monica, California, pp. 50-59.

[16] London, R. L., 1970. Proof of algorithms: a new kind of certification (certification of algorithm 245 TREESORT Comm. ACM, Vol. 13, No. 6, June 1970, pp. 371-373.

[17] London, Ralph L., and Musser, David R. 1974. The application of a symbolic mathematical system to program verification, Proceedings of the National ACM Conference, San Diego.

[18] Moore, J. Strother 1975. Introducing iteration into the pure LISP theorem prover, Xerox Palo Alto Research Center, CSL-74-3.
Ragland, L. C., 1973. A verified program verifier, Ph.D. thesis, University of Texas at Austin.

[19] Parnas, D. L., 1972, On the criteria to be used in decomposing systems into modules, Comm. ACM, Vol. 15, No. 12, December 1972, pp. 1053-1058.

[20] Reboh, Rene, and Sacerdoti, Earl, 1973. A preliminary QLISP manual, Stanford Research Institute, Aritificial Intelligence Center, Technical Note 81.

[21] Schaefer, Marvin 1973. A mathematical theory of global program optimization, Prentice Hall, Englewood, Cliffs, N. J.

[22] Schorre, D. V., 1974. Proof of correctness, EASCON '74. Washington, D.C. pp. 128-132.

[23] Sites, Richard L. 1974. Proving that computer programs terminate cleanly, Computer Science Department, Stanford University STAN-CS-74-418, May 1974.

[24] von Henke, F. W. and Luckham, D. C. 1975. A methodology for verifying programs, International Conference on Reliable Software.

[25] Waldinger, R. J., and Levit, K. N., 1973. Reasoning about programs, Stanford Research Institute, Artificial Intelligence Center, Technical Note 86.

[26] Wegbreit, Ben, 1974. The synthesis of loop predicates, Comm. ACM, Vol. 17, No. 2, February 1974, pp. 102-112.

[27] Wirth, N., 1970. The programming language PASCAL, Acta Informatica 1, pp. 35-63.

[28] Wulf, William A., 1974. ALPHARD: toward a language to support structured programs, Computer Science Department, Carnegie-Mellon University, April 30, 1974.

[29] ----------. CWIC: a compiler for writing and implementing compilers, System Development Corporation, TM-4662.

[30] Zislis, Paul 1975. Semantic decomposition of computer programs: an aid to program testing, Acta Information 4, pp. 245-269.

March 1976

AN EXAMPLE OF HIERARCHICAL DESIGN AND PROOF*

By: Jay M. Spitzen, Karl N. Levitt and Lawrence Robinson

ABSTRACT:

Hierarchical programming is being increasingly recognized as helpful in the construction of large programs. Users of hierarchical techniques claim or predict substantial increases in productivity and in the reliability of the programs produced. In this paper we describe a formal method for hierarchical program specification, implementation, and proof. We apply this method to a significant list processing problem and also discuss a number of extensions to current programming languages that ease hierarchical program design and proof.

*The research described here has been supported by the National Science Foundation under Grant DCR74-18661; the Air Force Office of Scientific Research under Contract F44620-73-C-0068; and the Office of Naval Research under Contract N00014-75-C-0816.

1. INTRODUCTION

Hierarchical programming is being increasingly recognized as helpful in the construction of large programs. Users of hierarchical techniques claim or predict substantial increases in productivity and in the reliability of the programs produced. In this paper we describe a formal method for hierarchical program specification, implementation, and proof. We apply this method to a significant list processing problem and also discuss a number of extensions to current programming languages that ease hierarchical program design and proof. The reader may consult [16] for a more detailed explanation of the method and [17] for a presentation of its application in designing a secure operating system.

Most contemporary programming languages provide some facilities for creating structured programs, for example, subroutines, conditional statements, and blocks, that can help to decompose a programming task into a set of subtasks. It is shown in [6] that the proof of a hierarchicallly structured program can be divided into showing that the parts satisfy their specifications and then, using just the specifications of the parts, proving the correctness of the whole program. Of course, this process can be repeated if the parts are similarly decomposable into subparts. (Note that the principle of recursion induction [9] asserts that this technique applies to recursive procedures as well.)

Syntactically structuring program text, so that concise mathematical specifications of textual fragments can avoid irrelevant implementation details (such as the values of temporary variables), is certainly a good idea. Unfortunately, with pure textual abstraction the specification of a fragment must state the relation between the fragment's inputs and outputs in terms of the concrete data structures of the language. This is unfortunate since, for example, a manipulation of a binary tree has to be specified in terms of the representation of the tree, perhaps an array. The design and proof process is surely facilitated if such manipulations are specified in terms of the primitive operations meaningful for trees--inserting nodes, accessing designated descendants of a node, etc. The general principle here is that implementation ought to be separated from specification.

Moreover, in a program organized as a set of procedures, there is often data that must be shared by several procedures. If such a group of procedures is correct, then surely each of them maintains shared data correctly. But, even if authorized procedures do correctly maintain shared data, in most languages there is no easy way to preclude other access to the data; erroneous modification of the common data by other parts of a large program remains possible. Thus, as Morris points out [12], "...we are only interested in those properties of a subprogram which are invariant over all the possible program texts into which the subprogram might be inserted." In the absence of constructs that forbid erroneous external modification, these invariant properties must depend on the validity of the common data. If erroneous external modification is impossible, then the difficulty of proving a small part of a large program is a function of the size of the part rather than of the size of the whole.

A number of modern programming languages, including Simula [1], Clu [7], EL1 [18], and Alphard [20], either provide (or, in the cases of Simula and EL1, can be easily extended to provide) constructs that limit access to common data in a practical and elegant manner. The common feature of these languages is provision for abstract data types. (We will also use the Simula term class.) The essence of the class is that it allows the collection of designated class data and class procedures into a single unit so that (in a modification of Simula to be described) it is possible to ensure that the class data are modified only by the class procedures. A Simula class definition defines the form of the members of the class--the class instances. Each instance may be viewed as an abstract sequential machine: the values of the local data of the class give its state; the class definition determines the initial state; and the invocations of class procedures cause state transitions. For example, an element of the class of binary search trees might have initial internal data representing an empty tree and procedures to insert new nodes and return the contents of existing nodes. An attractive feature of this organization is that the user manipulates class instances without access to or concern about their internal representation (except as provided indirectly through the class procedures).

In a hierarchy of class instances (also known as abstract machines), the machines form the nodes of a tree. The root machine solves a particular problem and the sons of each node are machines in terms of which the parent machine can be realized. The leaf nodes are primitive machines that are directly realized by some existing hardware or programming language implementation.

We have developed a method [16,17] that exploits the properties of such a hierarchy to enhance provability. A hierarchy of abstract machines is selected that solves a particular problem. Instead of directly implementing the machines, we first formally specify them as Parnas modules [14,15]. Next, for each nonleaf machine M, we write a representation function from a subset of the Cartesian product of the state spaces of M's sons onto the state space of M. Finally, we write abstract programs that implement the operations of each machine by suitable calls on the operations of its sons. It is then possible to prove the correctness of these programs with respect to the specifications of the machine, the specifications of its sons, and the corresponding representation function.

The proof of correctness of a complex program can be made practical by this approach. The hierarchy can be structured so that the abstract programs and the specifications are all quite concise. Thus, proving that a large program satisfies certain complex specifications reduces to proving that several small programs satisfy simpler specifications.

In Section 2, we present the formalism that we use for specification, implementation, and proof. In Section 3, we introduce a programming problem and specify a hierarchy of machines that gives a solution. This problem, solved by Deutsch [3], is to efficiently maintain (i.e., create and access) lists so that no two are isomorphic. In Section 4, we give representation functions and abstract programs for this hierarchy and sketch part of the proof of correctness. We also present a modification of Simula that serves as the basis of the implementation. We use Simula in a dual role: to provide the

syntax and primitive operations whereby an abstract machine is implemented by other machines; and as the language whose capabilities are the leaves of the hierarchy. In Section 5, we use the specifications of the abstract machines that are directly visible to the user to prove some results relating the functions available at this user interface. In Section 6, we present some concluding remarks.

2. DESIGN AND PROOF METHOD

We view program construction as consisting of the following stages:

(1) Selecting a hierarchy of abstract machines.

(2) Specifying each machine as a Parnas module. (Note that a single specification may describe each of a number of identically behaving machines.)

(3) Representing the states of each nonleaf machine in terms of the states of the machines directly below it.

(4) Implementing the initialization and operations of each nonleaf machine in terms of the operations of the machines directly below it.

(5) Writing each implementation as an executable program in a suitable programming language.

(This is a convenient way to present a finished result; in practice, trying to complete the process or to prove correctness often leads to revision of the original design.)

In the first two stages we design a hierarchy and describe each of its nodes as an abstract machine consisting of a state and various operations that can interrogate or change the state. Following [14], the state is defined by a set of value-returning functions called V-functions and the state-changing operations are defined by functions called O-functions and OV-functions. (O-functions serve only to change the state; OV-functions return a value as well.)

A V-function specification consists of:

An ASSERT section, which is an assertion characterizing the allowable argument values in terms of the abstract machine state.

An INITIAL section, which is an assertion constraining initial values of the function. (The INITIAL sections of a module, taken together, define the set of initial states of the abstract machine.)

An O- or OV-function specification consists of:

An ASSERT section which is like that of a V-function.

A set of expressions, called EFFECTS, characterizing the changed state due to a call on the function by means of the resulting V-function values. These expressions may allow several successor states (just as the INITIAL assertions may allow several initial states), but our proofs are independent of such ambiguities. Nondeterminism is permitted because it is a natural way to avoid "over specifying".

V-functions are either primitive or derived. The values of the primitive V-functions of a module determine its state, while the values of the derived V-functions are just a function of this state. Thus, instead of an INITIAL section, a derived V-function has a DERIVED section that gives its value in terms of primitive V-functions.

Finally, an OV-function specification is just like an O-function specification except that its EFFECTS section also constrains the value to be returned. Although OV-functions are not strictly necessary, their use makes specifications more concise and thus we use them freely.

The final stage of program construction, in this formal method, is implementation and, if desired, proof of correctness. Suppose machine M is a node in the hierarchy whose sons are the machines M1, ..., Mt. We refer to the functions of these machines by the following generic names: v for primitive V-functions; dv for derived V-functions; o for O-functions; and ov for OV-functions. A state of M is then an interpretation of its v's; a state of the abstract machine consisting of M1, ..., Mt (which we will call N) is an interpretation of the v's of all of the Mi's.

A formal implementation of M in terms of N then consists of:

(a) A representation function rep from a subset of the states of N onto the states of M. A state sn of N that is intended to model a state of M will be in the domain of rep and rep(sn) will be the modeled state. The use of rep is an aid to design and proof because it corresponds to the intuitive idea of composing a data structure out of simpler structures (cf. [5]).

(b) An initialization program INIT written in terms of N. INIT should achieve a state of N that models an initial state of M.

(c) A program of machine N, denoted impl(f,x), for each possible invocation f(x) of a function of M (where x is the n-tuple of arguments [x1, ..., xn]). If f is a v, dv, or ov, this program must return a result denoted impl*(f,x)),

Given a formal implementation, we wish to provide the facilities of M where only the facilities of N are primitively available. (Of course the process may be repeated to implement the components Mi of N in terms of simpler abstract machines.) With a formal implementation, the operations of M are obtained as follows:

(1) The components of N are initialized.

(2) INIT is executed.

(3) Then, for each call f(x) of a function of M, impl(f,x) is executed and, if appropriate, the result impl*(f,x) is returned.

We naturally intend that a formal implementation be correct. It should be noted that a major objective of the formalism we have described is to increase reliability of the programs that result by partitioning their construction into separate stages and by making their structure clear and explicit. Thus the importance of formal proof for establishing correctness is reduced. For the same reasons, however, if proof is desired, for example, for particularly critical modules of a system, it is much easier to achieve than it would be in the absence of careful structuring. Specifically, suppose the functions of N are available (either in underlying hardware or in software) and correctly implemented. Then we need sufficient conditions on INIT, rep, impl, and impl* to guarantee that the implementations of the functions of M satisfy the specifications of these functions. These conditions are as follows:

(C1) Suppose that T is an initial state of N and that state T' results from executing INIT beginning in state T. Then we require that T' $\in$ domain(rep) and that rep(T') is an initial state of M.

(C2) C1 guarantees that the states of M and N are initially "in step". It is also necessary that, whenever the states change, they remain in step. More precisely suppose that S=rep(T) and that impl(f,x), executed in N from state T, can yield state T' of N. (Recall the possibility of nondeterministic specifications for f.) Then we require that rep(T')=S', where S' is a possible result of the invocation f(x) from state S in M. Figure 1 describes this situation by showing states as nodes of a diagram and state transitions as directed arcs. C2 asserts that this diagram is commutative.

(C3) If S, T, and T' are as in (C2) and f is a v or dv, then rep(T)=rep(T'). That is, we do not allow the implementation of a V-function to change the state that is represented in the high-level machine.

(C4) The value that is returned by a V-function is implicit in the state of the abstract machine. The value to be returned by an OV-function is constrained by both the state of the abstract machine and the function's specification. In both cases, these constraints must be satisfied by the actual value that is returned by the corresponding abstract program--impl*(f,x). Specifically, if S, T, S', and T' are as in (C2) then:

(a) impl*(v,x) must be the value of v on argument x in S.

(b) The statement that dv(x), invoked in state S, returns impl*(dv,x), must be consistent with the derivation of dv in S.

(c) The statement that ov(x), invoked in state S, returns impl*(ov,x), must be consistent with the specification of (the result of) ov.

Note that proving C1-C4 for M and N, a pair of adjacent levels in a hierarchy, demonstrates that the correctness of the implementation of N implies the correctness of the implementation of M. Thus, if we carry out such proofs for all pairs of adjacent levels in a hierarchy, it follows that the correct operation of the lowest levels of the hierarchy (hardware or programming language system) implies the correctness of the entire implementation.

3. SPECIFICATION FOR MODULES OF THE EXAMPLE

Introduction

In this section we present a hierarchy of abstract machines to solve the problem of maintaining a class of unique lists--lists such that no two are structurally isomorphic. Thus the attempt to create one of these lists yields an old cell if a suitable cell exists. [As an example of the utility of such a facility, suppose we save the property SIMPLIFIES-TO-ZERO in some table under--as key--the address of the list (SUBTRACT x x). If we subsequently independently create a conventional list of the same form, it will have a different address and the property will not be retrieved. But if both are unique then their addresses will be the same so that the property can be looked up successfully.] Naturally, we want the check of existing cells to be efficient. We use a particularly effective method due to Deutsch, which he introduced in his verification system [3] to associate properties with arbitrary symbolic expressions.

We begin by explaining our notation. First, note that when we wish to refer to both an old and a modified machine state, we write 'v (x) to denote the result of this call in the old state and v(x) to denote the value in the new state. (Here x denotes a general argument n-tuple.) We omit from EFFECTS sections facts about the new state that were true in the former state.

Thus, if for some V-function v and argument x, there is no reference in an EFFECTS section to v(x), it is implicit that v(x) = 'v'(x).

Each function can have input arguments and--for V- and OV-functions--a single output argument, used to refer to the value returned. (The name of this result follows the semi-colon in the function header.) The type of an argument associated with a function is defined by the letter used for the argument in the header of the function specification. In the case of the CONS module the following types and their defining letters are used: b:Boolean, c:list cell, and x:any (i.e., unrestricted). Booleans are primitive in our language, but we specify the semantics of list cells. (We view the question of how best to use more complex data-types, such as those of EL1 [18], as worthy of study but do not address it here.) For OV- and derived V-functions the value given to the output argument, as a result of a call on the function, is generally characterized by the formulas in EFFECTS or DERIVED.

Specification of CONS

Clearly a facility is needed to maintain conventional lists. This is specified as an abstract machine, called the CONS module, that provides the following primitive list-processing functions:

consp(x;b)--returns the value TRUE if x is a previously created list.

car(x;x1)--returns the head of x. The precondition is consp(x), that is, x is a previously created list. The INITIAL values of car are unconstrained, since the precondition precludes arguments other than existing lists and there are no such values initially.

cdr(x;x1)--returns the tail of x. The preconditions and INITIAL values are the same as those of car.

cons(x1,x2;c)--returns a new list c, whose car is x1 and whose cdr is x2. Note that we assume an infinite supply of new values to be used as cons cells--a call may return any value (such as the address of a previously free cell) with suitable car, cdr, and consp.

equal(x1,x2;b)--returns TRUE if and only if x1 and x2 have the same list structure and identical non-list components. Equal is derivable from the other V-functions (consp, car, and cdr) and from the primitive function "=" which is TRUE only when its arguments are identical.

We specify this module as having primitive V-functions consp, car, and cdr; OV-function cons; and derived V-function equal. The formal specification is:

```
V-function:  consp(x;b)
  Initial:   b=FALSE

V-function:  car(x;x1)
  Initial:   TRUE
  Assert:    consp(x)

V-function:  cdr(x;x1)
  Initial:   TRUE
  Assert:    consp(x)

OV-function: cons(x1,x2;c)
  Assert:    TRUE
  Effects:   ~'consp'(c)
             consp(c)
             car(c)=x1
             cdr(c)=x2
```

```
V-function:  equal(x1,x2;b)
  Assert:    TRUE
  Derived:   if x1=x2 then b=TRUE else
             if consp(x1) and consp(x2)
                then b=(equal(car(x1),ca
                          and equal(cdr(x
                else b=FALSE
```

Observe that if a call of equal satisfies its guarantees that the arguments to any re assertions.

These specifications precisely describe processing operations: car, cdr, cons, cor LISP), = (which is called "eq" in LISP), ar original problem, however, since cons alway though the car and cdr of that cell are id cell. It is clear that we need additional me not happen. Thus we will specify search and keep track of existing list cells.

Specification of HASH-CONS

The HASH-CONS module is similar to the C in its list-creating mechanism. It is specif

```
V-function:  hconsp(x;b)
  Initial:   b=FALSE

V-function:  hcar(x;x1)
  Initial:   TRUE
  Assert:    hconsp(x)

V-function:  hcdr(x;x1)
  Initial:   TRUE
  Assert:    hconsp(x)

OV-function: hcons(x1,x2;c)
  Assert:    TRUE
  Effects:   Let A = {c1 : 'hconsp'(c1)
                       and 'hcar'(c1)=x1
             if A≠{} then c ∈ A
                     else ~'hconsp'(c)
                          hconsp(c)
                          hcar(c)=x1
                          hcdr(c)=x2
```

In the EFFECTS of hcons, the set A of existi will, in fact, always have at most one member Section 6, Theorem 1 below) and thus we use the specification to avoid the appearance of

The functions hconsp, hcar, and hcdr are, respectively, like c and cdr in CONS. The function hconsp returns TRUE if its argument i cons'ed list; hcar and hcdr return, respectively, the head and tai lists. The OV-function hcons, applied to arguments x1 and x2, retur cons'ed list with hcar x1 and hcdr x2--but this list is new o hcons'ed list already exists with these components. The reader ma the absence of an hequal, analogous to equal in the CONS module, b function is superfluous as we prove below.

In Section 6 we will prove several results about the CONS and machines as they appear to a top-level user. These proofs are based the specifications of CONS and HASH-CONS and the reader may prefer t them now, deferring the presentation and proof of the rest of the which is the subject of the remainder of this section, Section 4, a 5.

Specification of SEARCH

Our solution requires that all lists of the HASH-CONS maintained in a data base which can be (preferably rapidly) sea associative search, provided by our SEARCH module, is a convenient a for this purpose:

```
V-function:  get(x1;x)
  Initial:   x=NONE

OV-function: save(x1,x2;x)
  Effects:   if 'get'(x1)=NONE
                then x=get(x1)=x2
                else x='get'(x1)
```

Arbitrary values are saved and retrieved by means of a key function get does the retrieval--its argument is the key. Ini returns the distinguished default value NONE. The OV-function save, arguments x1 and x2, associates the value x2 with the key x1. Howe value has been previously saved with key x1, then that value [g returned and the data base is not changed. This dual use of save i because, in our application, once an entry is associated with a key, that correspondence never changes.

Specification of VARIABLE

Finally we need a VARIABLE module to maintain a single value, the distinguished value NONE. Its specification is:

```
V-function:  load(;x)
  Initial:   x=NONE
```

O-function: store(x)
Effects: load()=x

The V-function load returns the saved value and the O-function makes x the saved value. (This very simple module serves a m purpose in the hierarchy: we implement SEARCH in terms of CONS a Section 4, but could not implement it with CONS alone.)

Summary

We can now describe the organization of these four module: the unique list maintenance problem. It is given in Figure 2. HASH-CONS to create unique lists. He can also create conventi calling the (highest level) CONS machine. HASH-CONS is impl CONS, SEARCH and VARIABLE. As we explain in the next section, machines are used in the implementation, corresponding to the between HASH-CONS and SEARCH in Figure 2. Finally, each SEARC implemented by a VARIABLE machine and a CONS machine.

4. FORMAL IMPLEMENTATIONS

Introduction

In this section we present formal implementations for specified in Section 3. There are four such implementations: terms of CONS, SEARCH, and VARIABLE; SEARCH in terms of CONS CONS in terms of Simula; and VARIABLE in terms of Simula. We als proof of correctness of the HASH-CONS implementation.

Formal Implementation and Proof of HASH-CONS

First we describe the implementation of HASH-CONS. The stat CONS virtual machine M1 is defined by interpretations of the functions hconsp, hcar, and hcdr. We wish to represent such a st of a state of a virtual machine M2 for CONS, SEARCH, and VARIABLE is defined by the interpretations of the primitive V-functions cc cdr (of CONS), get (of SEARCH), and load (of VARIABLE). Thus, description of a formal implementation given in Section 2, we mu INIT program on M2, a rep function from some subset of the states states of M1, and implementations (by means of programs on M2) functions of M1.

To remember the set of known list cells, we use multiple SEA Each is a separate abstract machine whose state is independent of the others. We adopt the notation of Simula and write "s.f(x)" t

call of function f of machine s on argument x. The set of
consists of a distinguished machine and a dynamically varying
machines. The distinguished machine (recorded in VARIABL
correspondence between forms and the secondary machines
machines map from forms to forms. In particular, if c is a
then a retrieval from the distinguished SEARCH, with key h
SEARCH table s such that retrieval from s with key hcar(c)
two-level retrieval system is illustrated in Figure 3.
seventh list (a (b c) c) has the sixth list ((b c) c) as its
with key 6 in the primary SEARCH table gives a secondary tabl
key a--the hcar of 7--we find 7.] Furthermore, only hcons'ed
retrievable from this two-level structure. We make this re
by defining the domain of rep as the set of lower level state

```
(∀c,xa,xb)(load().get(xb)=NONE
            or (load().get(xb).get(xa)=c≠NONE
                ≡ (consp(x) and car(c)=xa
                            and cdr(c)=xb)))
```

The cells c satisfying this formula are exactly the lower-lev
HASH-CONS. Although the other lower-level cells are not
existence does not affect the correctness of the implementat
known cells is:

```
SD = {c : load().get(cdr(c)).get(car(c))=c}.
```

Let T be a state of M2 in domain(rep), defined by its
get, and load. We specify the represented state rep(T) by
hcdr, and hconsp:

hcar = car, restricted to the subdomain SD;

hcdr = cdr, restricted to the subdomain SD; and

```
hconsp(x) = consp(x)
              and load().get(cdr(x))≠NONE
              and load().get(cdr(x)).get(car(x))=x.
```

Thus hconsp is TRUE precisely for elements of SD.

For INIT we want a program that saves in VARIABLE a n
initialized SEARCH machine. Adapting Simula's notation, we w

```
store(new SEARCH).
```

Following the execution of INIT, there are no hcons'ed cells.

Finally, we implement the functions of M1. hconsp(x) is

```
consp(x) and load().get(cdr(x))≠NONE
         and load().get(cdr(x)).get(car(x))=x;
```

hcar(x) is implemented by:

```
assert hconsp(x);
car(x);
```

hcdr(x) is implemented by:

```
assert hconsp(x);
cdr(x);
```

and hcons(xa,xb) is implemented by:

```
load().save(xb, new SEARCH).save(xa, cons(xa,xb)).
```

The expression e=load().save(xb, new SEARCH) yields the secondary SEARCH table (either existing or newly created) in which the cell we seek should be recorded. Thus e.save(xa, cons(xa,xb)) finds that cell if it exists, or, failing that, records a suitable new cell in the table. Although this program may appear to be profligate in its creation of list cells and SEARCH tables, in fact the specification of save guarantees that previously stored entries are not overwritten, and the executable Simula version of save, given below, will be such that the second argument is not evaluated unless it is needed.

We now prove the correctness of this formal implementation. As we explained in Section 2, the proof requires verifying four conditions C1, C2, C3, and C4 as follows:

(C1) INIT is "store(new SEARCH)". Its execution from the initial state T of M2 yields a state T' in which load() is a new SEARCH and hence such that, for any x, load().get(x)=NONE. Clearly T' is in domain(rep) and rep(T') is a state of M1 in which the domains of hcar, hcdr, and hconsp are all empty, that is, the initial state of M1 as required.

(C2) The only state-changing function is hcons. Let impl(hcons, [xa,xb]) be executed from a state T in domain(rep), yielding a new state T'. If 'load'().'get'(xb)≠NONE and 'load'().'get'(xb).'get'(xa)≠NONE, then the implementation causes no state change in M2. Moreover, from the definition of rep, we deduce that a list cell c exists such that, in state S of M1, 'hconsp'(c), 'car'(c)=xa, and 'cdr'(c)=xb. Hence there is no state change in M1 or M2 and the commutativity of the diagram shown in Figure 1 is immediate. On the other hand, if either call of get does return NONE, then there is a state change in M2. However, it is then clear from the specifications and rep that no suitable HASH-CONS list cell existed in the old state S of the M1 (i.e., a cell with hcar xa and hcdr xb) and, moreover, that such a cell does exist in the new state S'. In this case it can be seen that the implementation of hcons changes T exactly by creating a list cell c such that load().get(fb).get(fa)=c. Hence rep(T) is S' and the diagram does commute.

(C3) It must be shown that the V-function implementations do not change the represented state of M1. But this is trivial since the implementing programs do not call any O-functions of M2 and hence cannot cause any state change in M2.

(C4) Here it is necessary to show that the values returned by the implementing programs on M2 agree with the specifications of M1. This is

immediate since the same list cells are employed at both levels to represent HASH-CONS lists and since the values and domains of car, cdr, and consp are updated exactly in parallel with those of their counterparts hcar, hcdr, and hconsp.

This completes the proof of the HASH-CONS implementation.

Formal Implementation of SEARCH

Next we describe a formal implementation of SEARCH. The implementation by Deutsch actually uses two different implementations: a conventional hashtable for the primary SEARCH table, and LISP assoc lists (i.e., lists of cons cells) for the subordinate tables. In this paper we present only the second of these for both purposes; the use of both is discussed in the last part of this section.

The state of a virtual machine M3 for SEARCH is defined by its interpretation of the primitive get. We wish to implement this machine by means of a machine M4 with the facilities of CONS and VARIABLE. The central point of this implementation will be to use LISP assoc lists to represent the map from keys to values. Thus we define the predicate wff-assoc-list, which tests whether an assoc list is well-formed, by:

```
wff-assoc-list(x) =
     x=NONE or (consp(x) and consp(car(x))
                         and wff-assoc-list(cdr(x))).
```

The domain of rep is now the set of states of M4 such that:

```
wff-assoc-list(load())
```

That is, the represented states are those such that the value in VARIABLE is a well-formed assoc list. For a state T in this domain, we define the represented state of M3 to be the state such that, for any x1:

```
get(x1) = getx(x1,load())
```

where the definition of getx is:

```
getx(x1,x) =
      if x=NONE then NONE else
      if car(car(x))=x1 then cdr(car(x))
                        else getx(x1,cdr(x)).
```

The specified initial state of M4 correctly models the initial state of M3; hence INIT is the empty program.

Finally, we implement the two functions of M3. The OV-function save is implemented by:

```
save(x1,x2) =
     if getx(x1,load())=NONE
        then store(cons(cons(x1,x2),load()));
     getx(x1,load());
```

and the V-function get is implemented by:

```
get(x1) = getx(x1,load()).
```

Thus we retrieve values by considering in order the elements of the linked list in which keys and values are stored.

Extensions of Simula

We want to describe the implementations of CONS and VARIABLE in terms of an underlying Simula machine. For the implementations described above, we have been able to refer to formal specifications for the abstract machines involved; here we must instead rely on informal descriptions of Simula, since a formal specification of the language is outside the scope of this paper. (The reader should consult [22] for an example of axiomatic specification of a programming language.) It is important to note that the correctness of the complete implementation depends upon the correctness of the underlying Simula system.

Simula is essentially an extension of Algol 60. The primary addition is the class construction. The definition of a class C is syntactically similar to an Algol block and has its own local variables and local procedures. It is activated by the call "new C" whose value is a new instance of the class; this value, x, satisfies the assertion "x is C". The significance of the class construct is that, unlike Algol, where the completion of a block or procedure activation results in the discarding of the local environment of the activation, this environment survives the initial activation in Simula and thus remains accessible by means of the "dot" notation used above. (Thus class instances may be used to model the abstract machines discussed above.) The interested reader should consult [1,2] for descriptions of standard Simula; we concentrate here on a number of modifications that we have adopted.

We begin with the assignment and equality operators. Although Simula distinguishes pointer (or reference) values from nonpointer values and uses different assignment operators for the two types, we omit this distinction and employ the single operator ':=' for both. The basic equality test, denoted by the infix =, should be taken to mean that its arguments are identical. For primitive objects (e.g., integers or Booleans) this is the conventional notion. For two compound objects (e.g., arrays), this implies that both arguments of the test refer to the same object--identical contents do not suffice. And for pointers (which in Simula include all class instances), the same object must be pointed to for the test to succeed. Thus for the purpose of proof, = is an identity relation: it is an equivalence relation such that equal expressions may be substituted for each other. (Note that this is not the case for the derived predicate "equal" on forms, though it will turn out to hold for the corresponding predicate on forms of HASH-CONS.)

The next set of changes to Simula involve declarations and procedure calls. We augment declarations by allowing an initial value to be specified with the key word initial; if this option is not used, then the initial value is the default NONE. Actual parameters to procedures are called by first use as suggested by [11] (where the rule is denoted normal evaluation); the actual parameter expression is not evaluated until needed in the procedure body and the value obtained at this first use is used at subsequent references without reevaluating the expression. (This differs from call-by-value where the expression is always evaluated exactly once and from call-by-name where it is evaluated as often as it is referred to; with call-by-first-use the expression is evaluated at most once.) We allow procedure definitions to be augmented by assert statements that provide Boolean preconditions for a call of the procedure. In an implementation these preconditions might be formally proved for all calls prior to compilation, tested dynamically at run time, yielding exceptions when they failed, or ignored (at the user's peril). Finally, for class activations, we sometimes write "c(...)" instead of "new c(...)" for conciseness.

Our last set of changes relates to the protection facilities of Simula. The value of Simula for our application derives from the possibility of using a class instance to represent an abstract machine with accesses to the environment of the instance corresponding to interrogation or modification of the abstract machine state. Unfortunately, in standard Simula, access to these environments is unrestricted and thus it is difficult to capture the semantics of abstract machines such as those of Section 3. Several authors have suggested protection features for Simula to remedy this problem [5,13]; we adopt an analgam of these suggestions.

The object is to control access to the locals of class instances. We achieve this by forbidding access unless it is explicitly authorized by the class definition. If a local procedure p of a class is declared public then it may be invoked, using the "dot" notation x.p(arg1, ..., argn), on any instance x that is available at the point of invocation. Other local procedures of the class are considered hidden and are thus only callable from within the class body. Similar restrictions apply to local data of a class: writable locals may be arbitrarily accessed; readable locals may be read but not changed; and other locals may not be accessed outside the class body.

These protection facilities will suffice for the present paper; it is interesting to note, however, a more flexible facility that is needed in general. Suppose a hierarchy of classes is constructed so that class c1 creates local instances of c2 and occasionally returns these instances as the result of public procedure calls. To implement a desired data abstraction, one might want to allow the public procedures of the local c2 instances to be invocable within c1 but not by the creator of c1. (This restriction would be necessary in our examples if existing CONS lists were modifiable by destructive operations such as the LISP rplaca and rplacd [10]; it would then be important to guarantee that a user of HASH-CONS could not modify lists represented by CONS lists, even though such modification was possible by the original creator of a CONS list.) Alphard provides a suitable mechanism for implementing this protection [20]; the reader should consult [12] for an extensive analysis of related issues.

Implementations of CONS and VARIABLE

Using Simula, thus modified, we can implement CONS and render the implementations of SEARCH and HASH-CONS given above by executable programs. First, CONS is implemented by declaring:

```
class cons(car, cdr); readable any car, cdr;
  comment The body of cons is null;
```

(Note the underscoring of reserved words.) The parameters of the definition allow initial values to be supplied when a new instance is created. Instances of the class cons then represent the individual cons cells of the specification; the OV-function call "cons(xa, xb)" is implemented by the Simula statement "new cons(xa, xb)"; for a cons instance c, the V-function calls "car(c)" and "cdr(c)" are implemented, respectively, as "c.car" and "c.cdr"; and the V-function call "consp(x)" is implemented by the Simula expression "x is cons". Note that we use the set of existing instances of the class cons to realize the CONS module specification. By contrast, the programs given below will realize HASH-CONS and SEARCH abstract machines by single instances of corresponding classes. (Since, in practice, there will be only a single underlying Simula machine, the implementation of consp just given is not strictly correct. The problem is that the Simula state contains three kinds of cons instances: those created explicitly by the user; those created by the implementation of HASH-CONS; and those created by the implementation of SEARCH. The implementation "x is cons" for "consp(x)" confuses these three. This technical difficulty may be resolved by defining identical but differently named classes--cons1, cons2, and con3--for the three uses. However, this would complicate the presentation unnecessarily and thus we retain the single cons.)

The VARIABLE module does not require a special Simula implementation--its facilities are primitive in the language, provided by declared variables of suitable scope.

SEARCH and HASH-CONS Revisited

The SEARCH implementation given above may be straightforwardly translated to the Simula class declaration:

```
class assocer;
begin
 any x;
 any public procedure save(x1,x2); any x1,x2;
  begin
     save := save2(x1, x);
     if save=NONE then
        begin x := cons(cons(x1, x2), x); save := x2 end
  end;
 any public procedure get(x1); any x1;
  get := save2(x1, x);
 any procedure save2(x1, x); any x1, x;
  save2 := if x=NONE then NONE else
           if x.car.car=x1 then x.car.cdr
              else save2(x1, x.cdr)
end;
```

whose local variable x realizes the VARIABLE of the formal implementation and whose public procedures are the required O- and V-functions of the module.

An important advantage of the design method presented here is that the correctness of the implementation of HASH-CONS is independent of the particular implementations of lower-level modules, so long as those modules satisfy their specifications. This fact can be applied in our example by using two different implementations of SEARCH: a conventional hashtable with open searching for the distinguished SEARCH table, and the assoc list version just given for the others. For a correct hashtable implementation, the reader is referred to [19]; here we omit the details and give only the structure of the implementation (which does satisfy the SEARCH specification). This is as follows:

```
class hasher;
begin
 ---
 assocer public procedure save(x1,x2);
    any x1; assocer x2;
    ---
 assocer public procedure get(x1); any x1;
    ---
 ---
end;
```

Finally, HASH-CONS is realized by another straightforward translation to Simula of a formal implementation, building on the previously defined classes hasher, assocer, and cons. The required definition is:

```
class hash-conser;
begin
 hasher a initial hasher();
 any public procedure hcar(x); cons x;
   hcar := x.car;
 any public procedure hcdr(x); cons x;
   hcdr := x.cdr;
 any public procedure hcons(x1, x2); any x1, x2;
   hcons := a.save(x2, assocer()).save(x1, cons(x1, x2));
 Boolean public procedure hconsp(x); any x;
   begin
     any temp;
     hconsp := x is cons
                 and (temp := a.get(x.cdr))≠NONE
                 and x=temp.get(x.car)
   end
end;
```

Note that call-by-first-use assures that the second arguments to save are evaluated only if they are needed.

5. SOME RESULTS ABOUT HASH-CONS AND CONS

Consider the interface available to a top-level user, as shown in Figure 2. A HASH-CONS machine provides unique list-processing and a CONS machine provides conventional list-processing. We wish to prove some properties of the lists that are obtained by the user of these machines. The first of these may be viewed as part of the assertion that HASH-CONS does indeed provide unique lists. That is, if two unique lists have the same hcar and hcdr, then they are identical. More precisely, we have:

Theorem 1. $(\forall c1)(\forall c2)$ hconsp(c1) and hconsp(c2)
and hcar(c1)=hcar(c2)
and hcdr(c1)=hcdr(c2)
-> c1=c2.

Proof. The theorem will be proved by induction on sequences of states of HASH-CONS. This method of proving properties of modules or protected classes, which we call generator induction, is discussed in [5,19,21]. We must prove the theorem for each initial state of HASH-CONS and for each state S' such that the theorem holds in some immediately preceding state S. So that it is clear, in each of the formulas that follows, which state is involved, we will concatenate state names and function names. For example, hcarS(c) denotes the hcar of c in state S.

The basis of the induction is immediate, since hconspI(c) is FALSE for any c and initial state I. Thus it suffices to assume the theorem holds in some state S and to deduce its validity in a successor state S' that results from the operation hcons(x1,x2) (since hcons is the machine's only state-changing operation). Suppose that c1 and c2 are lists such that hconspS'(c1),

hconspS'(c2), hcarS'(c1)=hcarS'(c2), and hcdrS'(c1)=hcdrS'(c2). We must show that c1=c2. This is an immediate consequence of the inductive assumption if hconspS(c1) and hconspS(c2) both hold so we may assume that either ~hconspS(c1) or ~hconspS(c2). Without loss of generality, assume that ~hconspS(c1). Then, according to the specification of hcons(c1, c2) and the fact that hconspS'(c1), it must be the case that c1=hconsS(x1, x2). We now consider two cases: ~hconspS(c2) or hconspS(c2).

In the case ~hconspS(c2), it follows as above from the definition of hconsS(x1, x2) and the fact that hconspS'(c1), that c2=hconsS(x1, x2), and hence that c1=c2 as required.

In the second case, hconspS(c2) together with the facts that hcarS'(c2)=hcarS'(c1)=x1 and hcdrS'(c2)=hcdrS'(c1)=x2, imply by the specification of hcons that S'=S. Consequently, the result that c1=c2 is immediate by the inductive assumption and the proof is complete.

Next, we extend this result to show that equal is not needed for HASH-CONS lists because its function is served by "=". In the proof, we will need a function hequal for HASH-CONS, which we have not yet defined; it is defined by analogy with CONS equal in the obvious way.

Theorem 2. (∀x1,x2)[hequal(x1, x2) -> f1=f2]

Proof. Note that only a single state occurs in this theorem and proof, so the special notation of Theorem 1 is not needed. If ~hconsp(x1), then the result is a straightforward consequence of the definition of hequal. To prove the result when hconsp(x1), note that hequal(x1, x2) implies that hconsp(x2). We proceed by structural induction [21], assuming the result true for hcar(x1) and for hcdr(x1) (and all x2). Then hequal(x1, x2) implies that hequal(hcar(x1), hcar(x2)) and hequal(hcdr(x1), hcdr(x2)); hence by the inductive assumption, hcar(x1)=hcar(x2) and hcdr(x1)=hcdr(x2). Theorem 1 now applies and yields the desired result.

Next we prove a theorem about a program that might be run by a top-level user of these machines user of these machines to translate conventional lists to unique lists. We claim that this may be done by the function hcopy defined as follows:

```
hcopy(x) := if consp(x)
              then hcons(hcopy(car(x)), hcopy(cdr(x)));
              else x;
```

The major result about hcopy is that if "equal" holds for any two forms, then "=" holds for their hcopies. To state this precisely, we define correct-hcopy(xa,xb) to be:

```
equal(xa,xb) -> hcopy(xa)=hcopy(xb).
```

We can then prove:

Theorem 3. (∀xa,xb)[correct-hcopy(xa,xb)].

This formula may appear ill-formed, since the effects and result of hcopy are contingent upon--and also modify--the state of HASH-CONS. A more precise statement is as follows. Suppose equal(xa,xb). Suppose that hcopy is applied to xa, beginning in some HASH-CONS state S1. This application terminates (since our lists are acyclic) yielding a value xa' and a state S2. Suppose, moreover, that S3 is any successor of S2, resulting from a series of state transitions of HASH-CONS, starting at S2. Finally, suppose that hcopy, applied to xb from state S3, yields value xb' and state S4. Then xa'=xb'.

We are going to prove the result about hcopy by induction, which would be very cumbersome if an inductive assumption of this rather lengthy form were required. Fortunately, HASH-CONS satisfies the powerful frame axiom that, if hconsp(x) holds in some state S, then it holds in every successor state. This axiom allows us, without risk of unsoundness, to omit further references to changing states and use the simple statement of the theorem.

Proof. First, suppose ~consp(xa). Then equal(xa, xb) implies that xa=xb. Also, hcopy(xa)=xa and hcopy(xb)=xb so that the desired result is immediate. Next suppose consp(xa). Then consp(xb). We proceed by simultaneous structural induction on xa and xb. That is, we assume

(I1) (∀x)correct-hcopy(s(xa),x)

(I2) (∀x)correct-hcopy(x,s(xb))

where s is either car or cdr. (Clearly, correct-hcopy is symmetric in its two arguments.) From equal(xa, xb) it follows that

(I3) equal(car(xa), car(xb))

(I4) equal(cdr(xa), cdr(xb))

Combining these results with I1 and I2 we obtain

(I5) hcopy(car(xa))=hcopy(car(xb))

(I6) hcopy(cdr(xa))=hcopy(cdr(xb))

We must prove that hcopy(xa)=hcopy(xb). This is done as follows:

```
hcopy(xa)=hcons(hcopy(car(xa)), hcopy(cdr(xa)))
          {by the definition of hcopy}
         =hcons(hcopy(car(xb)), hcopy(cdr(xb)))
          {by I5, I6}
         =hcopy(xb)
          {by the definition of hcopy}
```

6. CONCLUSION

The design and implementation of complex programs is presently expensive and time consuming. Moreover, the programs that result are often obscure and

unreliable. Yet the attempt to increase reliability by proving program correctness can substantially add to initial development costs if conventional design and implementation techniques are followed. In this paper, we have presented a formal discipline for constructing large programs and we have discussed a number of programming language features that make it easy to derive an executable program from a formal implementation. We believe that these techniques have great potential for making proof a useful tool. Careful hierarchical structuring in program design leads to cleanly structured programs and formal specification of program modules provides a precise guide to implementors (as well as to managers). Also, the partitioning that is inherent in the programs that result, as illustrated in our example, leads to corresponding partitioning in proofs about these program. Thus these techniques can enable the use of program proving as an effective technique in developing large and complex systems.

Acknowledgement. The idea of trying to prove Theorem 3 is due to S. German. We thank B. Elspas and R. Boyer for reading and commenting on drafts of this manuscript. The research described here has been supported by the National Science Foundation under Grant DCR74-18661; the Air Force Office of Scientific Research under Contract F44620-73-C-0068; and the Office of Naval Research under Contract N00014-75-C-0816.

REFERENCES

1. Dahl, O. J., Myhrhaug, B., and Nygaard, K. Common Base Language, S-22, Norwegian Computing Center, Oslo, Norway (Oct. 1970).

2. Dahl, O. J., Dijkstra, E. W., and Hoare, C. A. R. Structured Programming, (Academic Press, New York, N.Y., 1972).

3. Deutsch, L. P. An Interactive Program Verifier, Ph.D. Thesis, Dept. of Computer Science (University of California at Berkeley, 1973).

4. Good, D. I. Provable Programming, Proceedings of International Conference on Reliable Software, SIGPLAN Notices, 10, 6 (June 1975), pp. 411-419.

5. Hoare, C. A. R. Proof of Correctness of Data Representations, Acta Informatica, 1, 4 (1972), pp. 271-281.

6. Igarashi, S., London, R. L., and Luckham, D. C. Automatic Program Verification I: A Logical Basis and its Implementation, Acta Informatica, 1, 4 (1975), pp. 145-182.

7. Liskov, B. and Zilles, S. Programming with Abstract Data Types, Proceedings of ACM SIGPLAN Conference on Very High Level Languages, SIGPLAN Notices, 9, 4 (Apr. 1974), pp. 50-59.

8. Liskov, B. and Zilles, S. Specification Techniques for Data Abstraction, IEEE Trans. Software Engineering, SE-1, 1 (March 1975), pp. 7-19.

9. McCarthy, J. A Basis for a Mathemati
Computer Programming and Formal Syste
pp. 33-70 (North-Holland, Amsterdan,

10. McCarthy, J. et al. LISP 1.5 Progra
Press, Cambridge, Mass., 1962).

11. Manna, Z., Ness, S., Vuillemin, J. I
Properties of Programs, C. ACM, 16, 8

12. Morris, J. Protection in Programming
1973), pp. 15-21.

13. Palme, J. Protected Program Modules
National Defense, Stockholm, Sweden (

14. Parnas, D. L. A Technique for Softwa
Examples, C. ACM, 15, 5 (May 1972), p

15. Parnas, D. L. On the Criteria to be
Modules, C. ACM, 15, 12 (Dec. 1972),

16. Robinson, L. and Levitt, K. N. Proo
Structured Programs, (submitted for p

17. Robinson, L. et al. On Attaining Re
Operating System, Proceedings of Inte
Software, SIGPLAN Notices, 10, 6 (Jun

18. Wegbreit, B. The Treatment of Data T
1974), pp. 251-164.

19. Wegbreit, B. and Spitzen, J. M. Pro
Structures, J. ACM, (to appear).

20. Wulf, W. A. ALPHARD: Toward a Langua
Computer Science Department, Carnegie

21. Burstall, R. M. Proving Properties o
Computer Journal, 12, 1 (Jan. 1969),

22. Hoare, C. A. R. and Wirth, N. An Ax
Programming Language PASCAL, Acta Inf

23. Morris, J. M. Types Are Not Sets, AC
Programming Languages, Boston, Mass.

:al Theory of Computation. In
ns, Braffort and Hirschberg (Eds.),
1963).

mmer's Manual, (The M. I. T.

nductive Methods for Proving
(Aug. 1973), pp. 491-502.

Languages, C. ACM, 16, 1 (Jan.

In Simula 67, Research Institute of
July 1973).

re Module Specification with
p. 330-336.

Used in Decomposing Systems into
pp. 1053-1058.

f Techniques for Hierarchically
ublication).

liable Software for a Secure
rnational Conference on Reliable
e 1975), pp. 267-284.

ypes in EL1, C. ACM, 17, 5 (May

ving Properties of Complex Data

ge to Support Structured Programs,
-Mellon University, (Apr. 1974).

f Programs by Structural Induction,
pp. 41-48.

iomatic Definition of the
ormatica, 2, 4 (1973), pp. 335-355.

M Symposium on Principles of
(Oct. 1973), pp. 120-124.

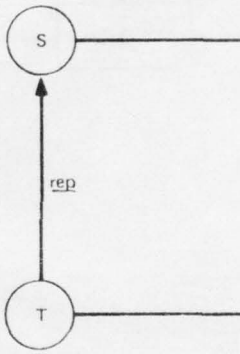
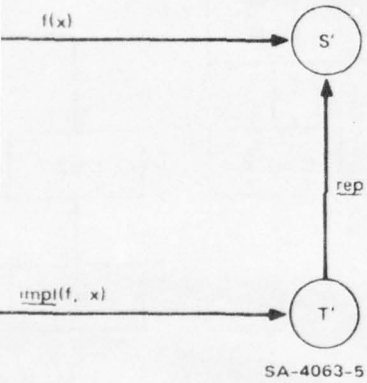


FIGURE 1 COMMUTA
FORMAL



SA-4063-5

TIVE DIAGRAM FOR A CORRECT
MPLEMENTATION

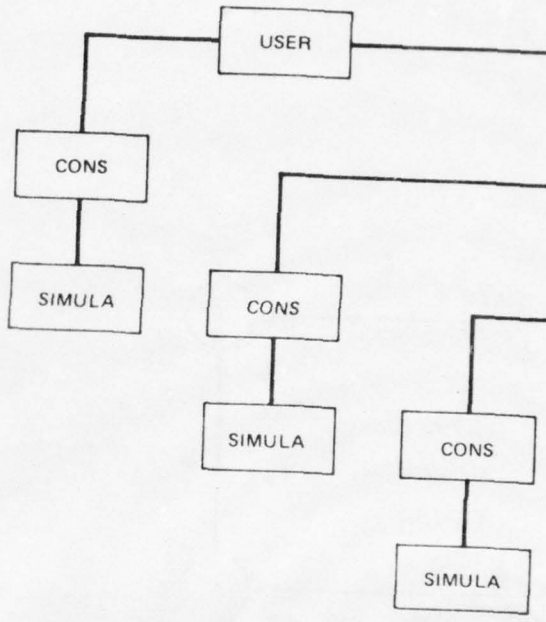
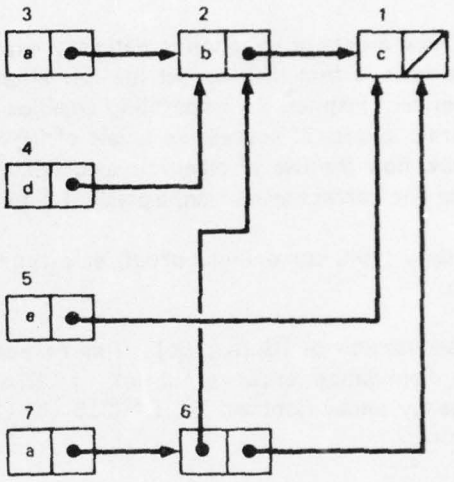


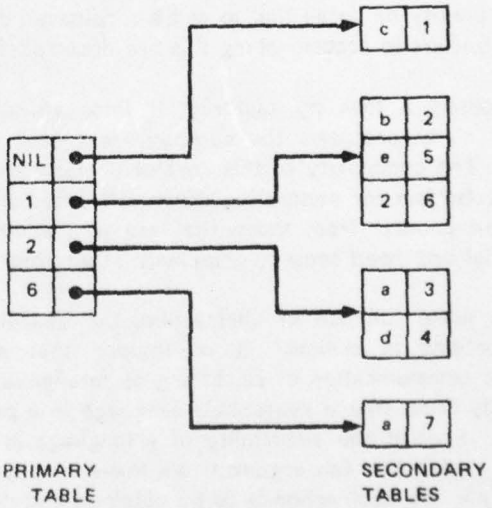
FIGURE 2 ORGANIZATION OF A HIERARCH
LIST PROCESSING

1. (c)
2. (b c)
3. (a b c)
4. (d b c)
5. (e c)
6. ((b c) c)
7. (a (b c) c)

(a) A SET OF SEVEN LIST STRUCTURES



(b) THE SAME LIST STRUCTURES IMPLEMENTED WITH SHARING OF COMMON CELLS



(c) THE SEARCH TABLES FOR THESE LISTS

FIGURE 3 DATA STRUCTURES USED BY HASH-CO

VERIFICATION OF DATA TYPES USING ALGEBRAIC AXIOMS

by

John V. Guttag*, Ellis Horowitz*, and David R. Musser**

Abstract: It is shown how a data abstraction is naturally expressed using algebraic axioms. The virtue of these axioms is that they permit the formal specification of a data type in a representation-independent manner. A moderately complex example is given which shows how to employ algebraic axioms at successive levels of implementation. The major thrust of the paper is to show how the use of algebraic axiomatizations can significantly improve the process of proving the correctness of an implementation of an abstract data type.

Key Words: abstract data type, correctness proof, data type, data structure, specification, software specification.

This is an abbreviated version of [Guttag76c]. This research was supported in part by the National Science Foundation under grant No. MCS76-06089 and by the Defense Research Projects Agency under Contract No. DAHC15 72 C 0308. The views expressed are those of the authors.

1. *Introduction*

The key problem in the design and validation of large software systems is reducing the amount of complexity or detail that must be considered at any one time. Two common and effective approaches to accomplishing this are decomposition and abstraction.

One decomposes a task by factoring it into two or more separable sub-tasks. Unfortunately, for many problems the separable sub-tasks are still too complex to be mastered *in toto*. The complexity of this sort of problem can be reduced via abstraction. By providing a mechanism for separating those attributes of an object or event that are relevant in a given context from those that are not, abstraction serves to reduce the amount of detail that one need come to grips with at any one time.

If one is to make full use of abstraction, the availability of a good notation for expressing abstractions is critical. It is obvious that a reasonable language is a prerequisite to the communication of something as intangible as an abstraction. It is less obvious, but equally true, that a reasonable language is a prerequisite to the creation of such abstractions. Even if the availability of a language is not necessary for the initial formulation of an abstraction (an argument we leave to psychologists and linguists), it is certainly necessary if the abstraction is to be retained and developed over any significant period of time.

---

* Computer Science Department, University of Southern California, Los Angeles, Ca. 90007

**USC Information Sciences Institute, 4676 Admiralty Way, Marina del Rey, Ca. 90291

A recent trend in programming is the development of abstract data types or data abstractions. In a data abstraction, a number of functional abstractions are grouped together. The clustered operations are related by the fact that they, and only they, operate on a particular class or type of object. Some typical data abstractions are: a symbol table, a priority queue, and a set.

In this paper we shall present a notation for describing data abstractions which we call algebraic axioms. Our experience indicates that the resulting specification of a data abstraction using algebraic axioms is both rigorous and easy to comprehend; see [Guttag76b]. However, the point we wish to stress in this paper is not design but the use of algebraic axioms for proofs of correctness. In Section 4 we show how this axiomatic technique can be profitably employed to prove the correctness of an implementation of a data abstraction. The strength of the technique is that it factors the proving process into distinct, manageable stages, and further, it simplifies the proof at each stage. In [Guttag76c] we discuss an automated system which processes an algebraic axiomatization of a data abstraction in such a way that correctness proofs can be carried out semi-automatically, and, in addition, programs may be tested before an implementation into a conventional programming language is achieved. This coupling of testing and correctness is a valuable by-product of the algebraic axiom approach and is a strong argument for its worth.

2. *Definitions and examples*

Rather than presenting formal definitions of abstract data types and related concepts, we prefer to give informal and (hopefully) intuitively appealing definitions and to illustrate the main ideas with a number of examples. We shall view a *data type* T as a class of values and a collection of operations on the values. If the properties of the operations are specified only by axioms, we call T an *abstract data type* or *a data abstraction*. An *implementation* of a data abstraction is an assignment of meaning to the values and operations in terms of the values and operations of another data type or set of data types. A *correct implementation* of a data abstraction is an implementation which is expressed in terms of correct data types and which satisfies the axioms.

An *algebraic axiom specification* of a data type T consists of a *syntactic* and a *semantic* description. The syntactic specification defines the names, domains and ranges of the operations of T. The semantic specification contains a set of axioms in the form of equations which relate the operations of T to each other. The term "algebraic" is appropriate because the values and axioms can be regarded as an abstract algebra. [Goguen75] and [Zilles75] have strongly emphasized the algebraic approach, developing the theory of abstract data types as an application of heterogeneous algebras. Implementations are treated under this approach as other algebras, and the problem of showing an implementation is correct is treated as one of showing the existence of a homomorphic mapping from one algebra to another. We shall in this paper de-emphasize the explicit use of algebraic terminology, preferring instead the terminology of programming. In spite of this difference in terminology, there are many similarities between our approach and the more purely algebraic approach. A significant technical difference which does exist between the two approaches is discussed in [Guttag76c].

The choice of a language in which to express the specifications is of importance. We must be able to express the relationships among the operations both precisely and

clearly. In addition, the specification language itself must be axiomatically defined to facilitate correctness proofs. We begin by assuming a base language with five primitives: functional composition, an equality relation (=), two distinct constants (TRUE and FALSE), and an unbounded supply of free variables. From these primitives one can construct an arbitrarily complex specification language; for once an operation has been defined in terms of the primitives, it may be added to the specification language. An IF-THEN-ELSE operation, for example, may be defined by the axioms:

IF-THEN-ELSE(TRUE,q,r) = q,
IF-THEN-ELSE(FALSE,q,r) = r.

We shall assume that the expression IF-THEN-ELSE(b,q,r), which we shall write as IF b THEN q ELSE r, is part of the specification language. We shall also assume the availability of infix Boolean operators such as $\wedge$, $\vee$, $\neg$, $\supset$, $\equiv$, etc. Finally, we allow for the conventional operations on integers: PLUS, MINUS, TIMES, DIV, MOD, and use the conventional infix operators when convenient.

2.1 *Stack Example*

One of the simplest examples of an abstract data type is the unbounded Stack type.

Figure 2.1 Stack Data Type

type Stack[elementtype: Type]

syntax

NEWSTACK → Stack,
PUSH(Stack,elementtype) → Stack,
POP(Stack) → Stack,
TOP(Stack) → elementtype ∪ {UNDEFINED},
ISNEW(Stack) → Boolean,
REPLACE(Stack,elementtype) → Stack.

semantics

declare stk:Stack, elm:elementtype;
POP(NEWSTACK) = NEWSTACK,
POP(PUSH(stk,elm)) = stk,
TOP(NEWSTACK) = UNDEFINED,
TOP(PUSH(stk,elm)) = elm,
ISNEW(NEWSTACK) = TRUE,
ISNEW(PUSH(stk,elm)) = FALSE,
REPLACE(stk,elm) = PUSH(POP(stk),elm).

In the example of Figure 2.1 we have defined a data type Stack with six operations via a syntactic specification of these operations, and a semantic specification which is a set of seven equations relating the operations. Certain notational conventions exhibited by this example will be used throughout. Operation names are written using all capital

letters. The name of a data type begins with a capital. In the equations the lowercase symbols are free variables ranging over the domains indicated, e.g., stk ranges over the Stack type. The symbol elementtype is a variable ranging over the set of types and elm ranges over elementtype. This says that we can have a Stack of any type of elements (but all must be of the same type); what we have defined is thus not a single type, but rather a *type schema*. The binding of elementtype to a particular type, e.g. Stack[Integer], reduces the schema to a specification of a single type. Using the syntactic specification of the operations, it can be checked that each of the expressions in the axiomatic equations is well-formed in the sense that each operator is applied to the correct number of arguments and each argument has the correct type.

The equations are statements of fact (axioms) relating the values which are created by the operations, e.g., the equation

TOP(PUSH(stk,elm)) = elm

means that for any Stack value stk and any elementtype value elm, the result of PUSH(stk,elm) is a Stack value, stk1, such that TOP(stk1) yields the value elm. In viewing the equations in this way, we are not required to give any particular interpretation to the values; the "useful" properties of the values can be derived solely from the relations determined by the axioms. Thus, in designing computer implementations of the operations, we are free to represent the values in many different ways.

An implementation of the Stack data type which is commonly used is an implementation in terms of an (Array, Integer) pair. Each Stack value is represented by a structure with two components - an array, whose components are of type elementtype, and an integer indicating the position in the array of the top element of the stack. The specifications for an Array data type are given in Figure 2.2.

Figure 2.2 Array Data Type

type Array[domaintype:Type,rangetype:Type]

syntax

NEWARRAY → Array,
ASSIGN(Array,domaintype,rangetype) → Array,
ACCESS(Array,domaintype) → rangetype.

semantics

declare arr:Array, dval,dval1:domaintype, rval:rangetype;
ACCESS(NEWARRAY,dval) = UNDEFINED,
ACCESS(ASSIGN(arr,dval,rval),dval1) =
IF dval = dval1 THEN rval ELSE ACCESS(arr,dval1).

ASSIGN(arr,t,elm) means the array identical to arr except possibly in the t-th position, where the value is elm [McCarthy63]. ACCESS(arr,t) returns the value in position t of the array arr.

The implementation of the Stack data type with (Array, Integer) pairs is given in Figure 2.3. We have divided the implementation into a *representation* part and a *programs* part. This corresponds to the division of the specification into a syntactic part and a semantic part.

Figure 2.3 An Implementation of the Stack Data Type with (Array,Integer)pairs

representation STAK(Array[Integer,elementtype],Integer) → Stack[elementtype],

programs

declare arr:Array, t:Integer, elm:elementtype;
NEWSTACK = STAK(NEWARRAY,0),
PUSH(STAK(arr,t),elm) = STAK(ASSIGN(arr,t+1,elm),t+1),
POP(STAK(arr,t)) = IF t=0 THEN STAK(arr,0) ELSE STAK(arr,t-1),
TOP(STAK(arr,t)) = ACCESS(arr,t),
ISNEW(STAK(arr,t)) = (t=0),
REPLACE(STAK(arr,t),elm) =
IF t=0 THEN STAK(ASSIGN(arr,1,elm),1)
ELSE STAK(ASSIGN(arr,t,elm),t).

In this paper the language used to express programs is the same as the language for expressing axioms. Though we recognize that a richer language is usually more desirable, we have chosen to restrict ourselves here for several reasons. Most importantly, the proof procedure described in Section 4 derives much of its simplicitly from the use of this restricted set of constructs. Since all conventional programming language features can be automatically translated into our basic set, see [Manna74], no real advantage is lost. We are able to avoid issues of language design and concentrate on how the basic command set can be axiomatized, used for correctness proofs, and used to synthesize implementations. The clarity of the basic language for axiomatizing data types has been shown in [Guttag76b] and [Horowitz76].

We intend that all of the operations be purely "functional" or "applicative," i.e., have no side effects. This can imply an unrealistic degree of inefficiency for implementations. In the Stack implementation of Figure 2.3, for example, the implementation of PUSH must involve copying the Array component as well as the Integer component of the Stack representation. The basic framework can be extended to permit specification of operations with side effects, so that the obvious efficient implementations are possible. However, since the exposition of our proof techniques is facilitated by this restriction, we will continue to assume no side effects in this paper, and refer the reader to [Guttag76c] for a discussion of the extensions required to remove this restriction.

The correctness of an implementation of a data type can be proved by showing that each axiom of the semantic specification is satisfied by the programs. As a particularly simple example of such a proof, consider the fourth Stack axiom. Assuming stk=STAK(arr,t),

TOP(PUSH(stk,elm)) = TOP(PUSH(STAK(arr,t),elm))
= TOP(STAK(ASSIGN(arr,t+1,elm),t+1))
= ACCESS(ASSIGN(arr,t+1,elm),t+1)
= elm.

The other Stack axioms can be shown to be satisfied in a similar manner, although not quite so straightforwardly. The complications that arise will be dealt with in Section 4, which discusses verification of implementations in detail.

2.2 *Programs as axioms and axioms as programs*

In the discussion of the implementation for the Stack data type, we described STAK(arr,t) as a pair whose first component is an Array and second component is an Integer, and viewed equations such as

TOP(STAK(arr,t)) = ACCESS(arr,t)

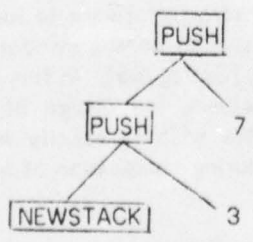
as definitions of programs for operating on the STAK pairs. Suppose, however, that we now view STAK as an *operation*, whose syntactic specification is STAK(Array[Integer,elementtype],Integer)→Stack[elementtype]. Then the above equation for TOP and the other program equations can be viewed as *axioms* which comprise a semantic specification for STAK. As an axiom, we would read the TOP equation as "if stk is the result of applying STAK to an Array arr and an Integer t, the value returned by TOP(stk) is ACCESS(arr,t)."

As an axiomatic specification of the Stack data type, the implementation of Figure 2.2 is inferior to the specification of Figure 2.1, in that it is not self-contained (it requires knowledge of properties of Arrays and Integers). We have called attention to the view of programs as axioms mainly because it suggests a *duality* between programs and axioms whose other half - *axioms as programs* - can be very fruitfully exploited.

We can, in fact, view the axioms of Figure 2.1 as programs, simply by regarding NEWSTACK and PUSH(stk,elm) as *trees* rather than operations. All structures built with NEWSTACK and PUSH can be pictured as trees. For example,

PUSH(PUSH(NEWSTACK,3),7)

can be diagrammed as



The Stack axioms can be viewed as defining operations which produce and access such tree structures:

NEWSTACK = |NEWSTACK|

PUSH(stk,elm) = |PUSH|
stk elm

POP(|NEWSTACK|) = |NEWSTACK|

POP(|PUSH|) = stk
stk elm

etc.

The two equations for POP together define POP as an operation which first checks which kind of node it is given and then proceeds accordingly. This is an example of a *direct implementation.*

Direct implementations are useful from a number of standpoints. In the first place, the concept of a direct implementation can serve as an aid to constructing specifications; i.e., one can try to write the semantic axioms so that they can serve as programs operating on tree structures. If this can be done, and one has a compiler for actually producing running implementations of such programs, then one can experiment with the operations, testing to a limited extent whether they have the properties intended. More importantly, one can also test high-level algorithms which are programmed in terms of the data type, before fixing upon a particular implementation of the data type. Thus, a true top-down design methodology can be achieved.

Many examples of algebraic axiomatizations of data types with explanations of their direct implementations appear in [Guttag76b]. [Guttag76c] contains a more detailed discussion of direct implementations and the related idea of "reduction systems."

3. *Symbol table example*

The Stack data type is too simple, in a number of respects, to properly illustrate the properties and uses of algebraic axiom specifications. The equations have too simple a form and the usable implementations are too straightforward to have "interesting" proofs of correctness. A much richer example is provided by the symbol table data type, which was first specified algebraically in [Guttag75], [Guttag76a]. In this example we deal with a common but non-trivial data structuring problem, the design of a symbol table for a compiler for a block-structured language. We wish to specify and implement a set of operations for maintaining the symbol table during compilation of a program. An informal specification of the operations might be:

INIT: allocate and initialize the symbol table.

ENTERBLOCK: prepare a new local naming scope.

ADDID: add an identifier and its attributes to the symbol table.

LEAVEBLOCK: discard entries from the most current scope and re-establish the next outer scope.

ISINBLOCK: has a specified identifier already been declared in this scope (used to check for duplicate declaration)?

RETRIEVE: return the attributes associated with the most local definition of a specified identifier.

The formal specification which we shall adopt is given in Figure 3.1:

Figure 3.1 The Symboltable Data Type

type Symboltable

syntax

INIT → Symboltable,
ENTERBLOCK(Symboltable) → Symboltable,
ADDID(Symboltable,Identifier,Attributelist) → Symboltable,
LEAVEBLOCK(Symboltable) → Symboltable,
ISINBLOCK(Symboltable,Identifier) → Boolean,
RETRIEVE(Symboltable,Identifier) → Attributelist ∪ {UNDEFINED}.

semantics

declare symtab:Symboltable, id,id1: Identifier, attrlist: Attributelist;

1) LEAVEBLOCK(INIT) = INIT,
2) LEAVEBLOCK(ENTERBLOCK(symtab)) = symtab,
3) LEAVEBLOCK(ADDID(symtab,id,attrlist)) = LEAVEBLOCK(symtab),
4) ISINBLOCK(INIT,id) = FALSE,
5) ISINBLOCK(ENTERBLOCK(symtab),id) = FALSE,
6) ISINBLOCK(ADDID(symtab,id,attrlist),id1) =
IF id = id1
THEN TRUE
ELSE ISINBLOCK(symtab,id1),
7) RETRIEVE(INIT,id) = UNDEFINED,
8) RETRIEVE(ENTERBLOCK(symtab),id) = RETRIEVE(symtab,id),
9) RETRIEVE(ADDID(symtab,id,attrlist),id1) =
IF id = id1
THEN attrlist
ELSE RETRIEVE(symtab,id1).

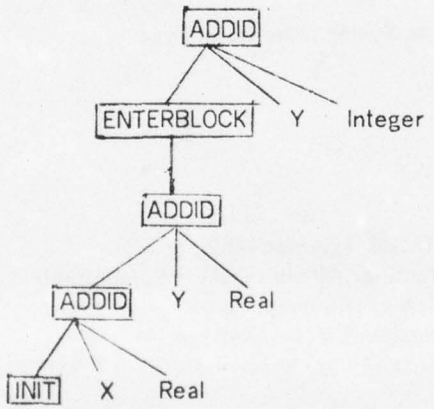
As an aid to understanding these axioms, it is useful to consider a direct implementation. We let the representation be trees of INIT, ENTERBLOCK, and ADDID nodes and use the full set of semantic axioms as programs. Then, for example, if this direct implementation is used by a compiler in processing the following program segment,

```
begin
        real X,Y;
          ...
        begin
                integer Y;
                  ...
        end
end
```

the symbol table

SYM=ADDID(ENTERBLOCK(ADDID(ADDID(INIT,X,Real),Y,Real)),Y,Integer)

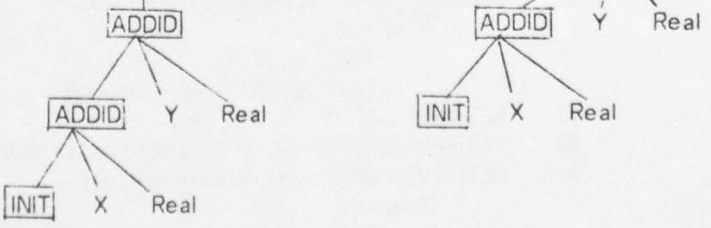
will be created within the innermost block. Diagrammed as a tree structure, this is:



Suppose now that we apply RETRIEVE to SYM and X. Simulating the RETRIEVE operation using the direct implementation, we have

RETRIEVE(SYM,X)

=[by axiom 9]=> RETRIEVE(ENTERBLOCK, X) =[by axiom 8]=> RETRIEVE(ADDID, X)



=[by axiom 9]=> RETRIEVE(ADDID, X) =[by axiom 9]=> Real.



If tree structure operations are implemented with reasonable efficiency, then this direct implementation could actually be used in a compiler to test the Symboltable specification more extensively, and, simultaneously, to test other components of the compiler. However, because this implementation requires potentially long searches for the RETRIEVE, ISINBLOCK, and LEAVEBLOCK operations, it is not very efficient.

The design of a hierarchically structured implementation of the Symboltable data type, using algebraic specifications of the data types employed at each level of the implementation, is carried out in detail in [Guttag76c]. Such a design was first presented in [Guttag75], where the lowest level of the implementation was expressed as a set of PL/1-like programs. The presentation in [Guttag76c] differs in that it uses the restricted set of language features described in Section 2. For brevity we present here only the top level of the implementation development.

If we ignored the complication introduced by block structure, a symbol table could be viewed abstractly as providing a mapping from identifiers to attribute lists. One way to handle block structure, especially suitable in a one pass compiler, is to have a stack of mappings, each mapping being from identifiers to attribute lists, with the top mapping on the stack corresponding to the current innermost block being processed. This is the method we have chosen in the implementation given in Figure 3.2.

Figure 3.2 An Implementation of the Symboltable Data Type with a Stack of Mappings

representation

```
SYMT(Stack[Mapping[Identifier,Attributelist]]) → Symboltable.
```

programs

```
declare stk:Stack, id:Identifier, attrlist:Attributelist;
INIT = SYMT(PUSH(NEWSTACK,NEWMAP)),
ENTERBLOCK(SYMT(stk)) = SYMT(PUSH(stk,NEWMAP)),
ADDID(SYMT(stk),id,attrlist) =
        SYMT(REPLACE(stk,DEFMAP(TOP(stk),id,attrlist))),
LEAVEBLOCK(SYMT(stk)) =
        IF ISNEW(POP(stk))
                THEN SYMT(REPLACE(stk,NEWMAP))
                ELSE SYMT(POP(stk)),
ISINBLOCK(SYMT(stk),id) = ISDEFINED(TOP(stk),id),
RETRIEVE(SYMT(stk),id) =
   IF ISNEW(stk)
        THEN UNDEFINED
        ELSE IF ISDEFINED(TOP(stk),id)
                THEN EVMAP(TOP(stk),id)
                ELSE RETRIEVE(SYMT(POP(stk)),id).
```

This implementation uses the operations of the Stack data type schema of Figure 2.1 and the Mapping data type schema of Figure 3.3. Note that we have bound the

AD-A033 625 ROME AIR DEVELOPMENT CENTER GRIFFISS AFB N Y F/G 9/2
PRESENTATIONS AT THE RADC/AR>A INVITATIONAL DOD/INDUSTRY CONFER--ETC(U)
AUG 76

UNCLASSIFIED NL

5 OF 5
AD
A033625



END

DATE
FILMED
2-77

parameters of the Stack and Mapping types. The Mapping data type is the same as the Array data type of Figure 2.2, except for the addition of an ISDEFINED operation.

Figure 3.3 Mapping Data Type

type Mapping[domaintype:Type,rangetype:Type]

syntax

NEWMAP → Mapping,
DEFMAP(Mapping,domaintype,rangetype) → Mapping,
EVMAP(Mapping,domaintype) → rangetype,
ISDEFINED(Mapping,domaintype) → Boolean.

semantics

declare map:Mapping, dval,dval1:domaintype, rval:rangetype;
EVMAP(NEWMAP,dval) = UNDEFINED,
EVMAP(DEFMAP(map,dval,rval),dval1) =
    IF dval = dval1 THEN rval ELSE EVMAP(map,dval1),
ISDEFINED(NEWMAP,dval1) = FALSE,
ISDEFINED(DEFMAP(map,dval,rval),dval1) =
    IF dval = dval1 THEN TRUE ELSE ISDEFINED(map,dval1).

The next level of the implementation discussed in [Guttag76c] contains the implementation of the Stack data type in terms of (Array,Integer) pairs, as discussed in the previous section, and an implementation of the Mapping data type in terms of a particular kind of hash table (an array containing lists of domain/range pairs). The discussion of verification methods in the next section does not rely on this material, and we omit it here.

4. *Proving correctness*

In this section we turn to the problem of proving correctness of implementations of data types. We shall continue to center the discussion around the example of the Symboltable data type. We shall show parts of the proof of correctness of the example implementation given in Section 3.

One of the most important aspects of the proof techniques which will be used to prove correctness of algebraically specified data types is that the proofs can be factored into levels corresponding to the levels of the implementation. To prove the correctness of a data type which is implemented in terms of other data types, we need to rely only on the axiomatic specifications of the other data types, *not* on their implementations. For example, as we will show in Section 4.2, to verify the top level of the implementation of the Symboltable data type, we use the semantic axioms for the Stack and Mapping data types, and ignore their implementations.

Another highly significant aspect of the use of axioms for other data types in the proof of an implementation is the computational nature of the proof steps: the axioms are used as rewrite rules and proofs proceed via a series of reductions. This aspect is also

illustrated in Section 4.2. The main implication of this is that to a large extent the proof process can be easily automated.

Two extremely important considerations in verifications of data type implementations are *data type invariants* and *equality interpretations.* Again, the algebraic axiom approach seems particulary well suited to dealing with these considerations. Section 4.3 discusses data type invariants, and we refer the reader to [Guttag76c] for a discussion of equality interpretations. Section 4.4 contains a brief discussion of an interactive system for verifying implementations of data types which is in use at the USC Information Sciences Institute.

4.1 *Formal deduction using the Boolean data type*

Although we will not be completely formal in the example proofs in the following sections, it is useful to discuss at this point the basis we have chosen for automating such proofs, since it fits very well with the overall approach of using algebraically specified data types. In fact, many of the formal deductions will be based on the following algebraic specification of the Boolean data type:

Figure 4.1 Boolean Data Type

type Boolean

syntax

TRUE ↛ Boolean,
FALSE → Boolean,
(IF Boolean THEN Boolean ELSE Boolean) → Boolean,
(Boolean ∧ Boolean) → Boolean,
(Boolean ∨ Boolean) → Boolean,
¬Boolean → Boolean,
(Boolean ⊃ Boolean) → Boolean,
(Boolean ≡ Boolean) → Boolean.

semantics

declare p,q,r:Boolean;
(IF TRUE THEN q ELSE r) = q,
(IF FALSE THEN q ELSE r) = r,
(p ∧ q) = IF p THEN q ELSE FALSE,
(p ∨ q) = IF p THEN TRUE ELSE q,
¬p = IF p THEN FALSE ELSE TRUE,
(p ⊃ q) = IF p THEN q ELSE TRUE,
(p ≡ q) = IF p THEN q ELSE ¬q,

In this specification each of the usual logical operators is related by an axiom to the IF-THEN-ELSE operator, which is axiomatized by the first two axioms relating it to TRUE and FALSE. The use of ∧, ∨, etc. in infix rather than prefix form is purely for

convenience. Although the above specification defines IF-THEN-ELSE only for Boolean operands, we assume that every data type T can use IF-THEN-ELSE with the syntactic specification

(IF Boolean THEN T ELSE T) → T

and the same axioms as the first two in the above specification.

We shall make frequent use of the following rewrite rules for IF-THEN-ELSE, which are theorems provable from the Boolean axioms:

1. [Repeated result rule] (IF p THEN q ELSE q) = q

2. [Redundant IF rule] (IF p THEN TRUE ELSE FALSE) = p

3. [IF-distribution rule] (IF(IF p THEN q ELSE r) THEN a ELSE b) =
   IF p THEN (IF q THEN a ELSE b) ELSE (IF r THEN a ELSE b)

4. [Logical substitution rules]
   (IF p THEN q[p] ELSE r) = (IF p THEN q[TRUE *for* p] ELSE r)
   (IF p THEN q ELSE r[p]) = (IF p THEN q ELSE r[FALSE *for* p]).

In the left hand side of a rule, "q[p]" means q must be an expression in which p occurs as a subexpression (possibly p=q). In the right hand side, "q[TRUE *for* p]" is the result of substituting TRUE for all occurrences of p in q. We require that p occur as a subexpression of q to limit applicability of the rule to those places where it will effect a change. Rules 1-4 are also theorems for IF-THEN-ELSE in other data types.

As noted in [McCarthy63] and [Boyer75], the Boolean axioms and rules 1-4 combine to yield a system for "simplifying" expressions of the propositional calculus, such as:

a) $A \wedge TRUE$,
b) $(A \vee B) \vee A$,
c) $((A \wedge B) \wedge A) \supset (B \wedge C)$
d) $((A \vee B) \supset C) \equiv ((A \supset C) \wedge (B \supset C))$

For example,

((A v B) v A) ==[by v axiom]==>
(IF (IF A THEN TRUE ELSE B) THEN TRUE ELSE A)

==[by IF-distribution rule]==>
(IF A THEN (IF TRUE THEN TRUE ELSE A)
ELSE (IF B THEN TRUE ELSE A))

==[by IF axiom]==>
(IF A THEN TRUE ELSE (IF B THEN TRUE ELSE A))

==[by logical substitution rule]==>
(IF A THEN TRUE ELSE (IF B THEN TRUE ELSE FALSE))

==[by Redundant IF rule]==> (IF A THEN TRUE ELSE B)

The result is equivalent to (A ∨ B) Similarly, a) can be reduced to A, c) can be reduced to a form equivalent to (A ∧ B) ⊃ C, and d) can be reduced to TRUE.

In [Guttag76c] we in fact prove that the Boolean axioms and rules 1-4 are *complete* with respect to the propositional calculus, in that any valid expression in propositional calculus (i.e., provable by truth table) is reducible to TRUE by use of these rewrite rules. These rules form the basis for an automatic simplifier for logical expressions described in [Guttag76c].

It is, of course, necessary to go beyond propositional calculus and include deductive rules for equality and other operators. We will not go in to the details of the formal rules here, except to mention the following important rule:

5. [Case analysis rule] $f(a_1, \ldots, (\text{IF } p \text{ THEN } x \text{ ELSE } y), \ldots, a_n) =$
IF p THEN $f(a_1, \ldots, x, \ldots, a_n)$
ELSE $f(a_1, \ldots, y, \ldots, a_n)$
when f ≠ IF-THEN-ELSE.

This is a "second-order" rewrite rule and the rule applies to IF-THEN-ELSE expresssions in any operand position. The case in which f is IF-THEN-ELSE is already covered in part by the IF-distribution rule. Note that if f were permitted to be IF-THEN-ELSE then the expression IF a_1 THEN (IF p THEN x ELSE y) ELSE a_3 could be transformed to IF p THEN (IF a_1 THEN x ELSE a_3) ELSE (IF a_1 THEN y ELSE a_3) and the rule would apply again, leading to an infinite sequence of applications. An important application of this case analysis rule occurs when f is "=", e.g.

((IF p THEN x ELSE y) = z) ==[by Case analysis rule]==> (IF p THEN (x=z) ELSE (y=z)).

4.2 *Verification of one of the Symboltable axioms*

The basic proof technique for verifying an implementation of a data type is to show that each of the axiomatic specifications for the data type is satisfied, when the programs for the operations of the data type are substituted into the axioms. As our first example, consider the ninth axiom for the Symboltable data type:

RETRIEVE(ADDID(symtab,id,attrlist),id1) =
IF id = id1 THEN attrlist
ELSE RETRIEVE(symtab,id1). (4.2-1)

To get started, we assume there exists a stack stk such that symtab = SYMT(stk). (We will show in Section 4.3 how to verify this assumption.) Substituting, we get the verification condition

```
RETRIEVE(ADDID(SYMT(stk),id,attrlist),id1) =
            IF id = id1 THEN attrlist
                        ELSE RETRIEVE(SYMT(stk),id1).                    (4.2-2)
```

The goal now is to show that this equation is true using the programs for RETRIEVE and ADDID (Figure 3.1) and the axioms for the Stack and Mapping data types (Figures 2.1 and 3.2) as rewrite rules. We must also use the axioms and some theorems for the Boolean data type, as discussed in Section 4.1.

Working first on the left hand side of (4.2-2), we make the following reductions:

```
LHS ==[by ADDID program]==>
    RETRIEVE(SYMT(stk1),id1) where stk1 = REPLACE(stk,DEFMAP(TOP(stk),id,attrlist))

==[by RETRIEVE program]==>
 IF ISNEW(stk1)
     THEN UNDEFINED
     ELSE IF ISDEFINED(TOP(stk1),id1)
             THEN EVMAP(TOP(stk1),id1)
             ELSE RETRIEVE(SYMT(POP(stk1)),id1)

==[by REPLACE, ISNEW, POP and TOP axioms]==>
   IF ISDEFINED(DEFMAP(TOP(stk),id,attrlist),id1)
                THEN EVMAP(DEFMAP(TOP(stk),id,attrlist),id1)
                ELSE RETRIEVE(SYMT(POP(stk)),id1)

==[by ISDEFINED axiom]==>
   IF (IF id = id1 THEN TRUE ELSE ISDEFINED(TOP(stk),id1))
                THEN EVMAP(DEFMAP(TOP(stk),id,attrlist),id1)
                ELSE RETRIEVE(SYMT(POP(stk)),id1)

==[by IF-distribution Rule]==>
   IF id = id1
     THEN IF TRUE
             THEN EVMAP(DEFMAP(TOP(stk),id,attrlist),id1)
             ELSE RETRIEVE(SYMT(POP(stk)),id1)
     ELSE IF ISDEFINED(TOP(stk),id1)
             THEN EVMAP(DEFMAP(TOP(stk),id,attrlist),id1)
             ELSE RETRIEVE(SYMT(POP(stk)),id1)

==[by IF axiom]==>
   IF id = id1
     THEN EVMAP(DEFMAP(TOP(stk),id,attrlist),id1)
     ELSE IF ISDEFINED(TOP(stk),id1)
             THEN EVMAP(DEFMAP(TOP(stk),id,attrlist),id1)
             ELSE RETRIEVE(SYMT(POP(stk)),id1)

==[by EVMAP axiom]==>
   IF id = id1
        THEN (IF id = id1 THEN attrlist ELSE EVMAP(TOP(stk),id1))
        ELSE IF ISDEFINED(TOP(stk),id1)
```

```
                    THEN (IF id = id1 THEN attrlist ELSE EVMAP(TOP(stk),id1))
                    ELSE RETRIEVE(SYMT(POP(stk)),id1)

==[by Logical substitution rule]==>
   IF id = id1
         THEN (IF TRUE THEN attrlist ELSE EVMAP(TOP(stk),id1))
         ELSE IF ISDEFINED(TOP(stk),id1)
                    THEN (IF FALSE THEN attrlist ELSE EVMAP(TOP(stk),id1))
                    ELSE RETRIEVE(SYMT(POP(stk)),id1)

==[by IF axioms]==>
   IF id = id1
         THEN attrlist
         ELSE IF ISDEFINED(TOP(stk),id1)
                    THEN EVMAP(TOP(stk),id1)
                    ELSE RETRIEVE(SYMT(POP(stk)),id1)
```

Having already substituted once for RETRIEVE, we do not substitute again for the recursive call. The right hand side of (4.2-2) reduces to the same expression upon use of the program for RETRIEVE. Thus (4.2-2) has been shown to be true, and axiom (4.2-1) has been verified for the implementation.

The reason for showing the proof of the axiom in as much detail as we have is that we wish to make clear that each step of the proof is a straightforward application of a reduction rule. The only choice to be made in terms of "proof strategy" is which of several possible reductions to apply at the next step. It should be noted, however, that the use of the axioms as left-to-right rewrite rules can be combined with restrictions on the form of the axioms (discussed in [Guttag76c]) to preclude "infinite loops" in the proof process. Thus, assuming that the axioms are satisfiable, all possible reduction sequences will terminate with the same result [Rosen73]. The reduction process is, therefore, readily automated. The above proof was carried out fully automatically by the prototype data type verification system at USC Information Sciences Institute.

Verification of the axioms in the manner of the above proof actually only establishes *partial correctness* of the implementation, i.e., that if the programs terminate they give results which satisfy the axioms. Proof of termination must be done separately. However, implementations of data types are often simple enough that termination is obvious, and we shall not deal with the issue of formal proofs of termination in this paper.

4.3 *Use and verification of data type invariants*

Although the example axiom verification of the previous section was long and required the use of many different rewrite rules, it did not illustrate the use of *data type invariants*. Consider, for example, the second symbol table axiom:

LEAVEBLOCK(ENTERBLOCK(symtab)) = symtab.

Substituting

symtab = SYMT(stk) (4.3-1)

and applying all possible reductions (including the case analysis rule with f = "="), we obtain

IF ISNEW(stk) THEN (SYMT(PUSH(NEWSTACK,NEWMAP)) = SYMT(stk))
ELSE (SYMT(stk) = SYMT(stk)). (4.3-2)

We cannot reduce this equation to TRUE unless we know that

ISNEW(stk) = FALSE. (4.3-3)

To prove (4.3-3), we recall that stk is not just an arbitrarily chosen stack, but one which was assumed to be generated as a representation of a symbol table. If we examine the syntactic specification of the Symboltable data type, we see that the only operations which produce symbol tables as their output are INIT, ENTERBLOCK, ADDID, and LEAVEBLOCK. Examining the program for each of these operations, we see that INIT generates an initial stack, stk, for which (4.3-3) is true, and that if (4.3-3) is true of the stack representing the symbol table argument of any of the other operations, then it is true of the stack produced in the result. Therefore, (4.3-3) must be true of all stacks produced as representations of symbol tables by operations of the Symboltable data type.

The general principle being used in the above proof is that of *data type induction* (called "generator induction" in [Spitzen75] and [Wegbreit76]). Paraphrasing the discussion in [Spitzen75], p. 141, we suppose that a data type T has, according to its syntactic specification, exactly the operations $F_1, \ldots, F_t$ whose range is the set of values of T. Let P(x) be a property of values of type T. Then if the truth of P for arguments of type T of each F_i implies the truth of P for the results of calls of F_i allowed by the syntactic specification of T, then it follows that P is true of all values of the data type. Assuming strong type-checking, the validity of this rule follows by induction on the number of computation steps involving values of type T. As Spitzen and Wegbreit point out, the data type induction principle "is analog to the principle of complete induction over the integers. As with complete induction, one of the results which must be established is the base step, that P is true of the results of those primitives F with no arguments of type T." In the case of symbol tables, INIT is the only such primitive.

Let us examine the proof of (4.3-3) by data type induction a bit more carefully. We can regard a property P_1 that we wish to prove about symbol tables by data type induction as an operation on symbol tables with the syntactic specification

P_1(Symboltable) → Boolean,

and the semantic specifications

P_1(INIT)
P_1(symtab) ⊃ P_1(ENTERBLOCK(symtab))
P_1(symtab) ⊃ P_1(ADDID(symtab,id,attrlist)
P_1(symtab) ⊃ P_1(LEAVEBLOCK(symtab)) (4.3-4)

which can be generated automatically from the syntactic specification of the Symboltable data type. To prove (4.3-3), we let the interpretation of P_1 in terms of the implementation values be

$$P_1(SYMT(stk)) = (ISNEW(stk) = FALSE). \qquad (4.3\text{-}5)$$

We then prove each of the conditions in (4.3-4) under this interpretation of P_1 along with the implementation programs of the Symboltable operations. For example, using (4.3-1), the fourth condition becomes:

P_1(SYMT(stk)) ⊃ P_1(LEAVEBLOCK(SYMT(stk)))

==[by (4.3-5)and LEAVEBLOCK program]==>
(ISNEW(stk) = FALSE)
⊃ P_1(IF ISNEW(POP(stk)) THEN SYMT(REPLACE(stk,NEWMAP))
ELSE SYMT(POP(stk)))

==[by case analysis rule]==>
(ISNEW(stk) = FALSE)
⊃ (IF ISNEW(POP(stk)) THEN P_1(SYMT(REPLACE(stk,NEWMAP)))
ELSE P_1(SYMT(POP(stk))))

==[by (4.3-5)]==>
(ISNEW(stk) = FALSE)
⊃ (IF ISNEW(POP(stk)) THEN ISNEW(REPLACE(stk,NEWMAP)) = FALSE
ELSE ISNEW(POP(stk)) = FALSE)

==[by ISNEW axiom and logical substitution rule]==>
(ISNEW(stk) = FALSE)
⊃ (IF ISNEW(POP(stk)) THEN FALSE = FALSE
ELSE FALSE = FALSE

which reduces to TRUE with the application of the reflexive property of Boolean equality, the repeated result rule, and the ⊃ axiom.

A property P which is true of all values of a data type is called an *invariant of the data type.* If the proof requires interpretation of P in terms of the implementation, then it is called an *implementation invariant* (of the data type). Thus, (with the establishment of the other three conditions in (4.3-4)), we have almost completed showing that (4.3-3) is an implementation invariant of the Symboltable data type. What remains to be shown is that

$$P(symtab) = (\exists\ stk \in Stack,\ symtab = SYMT(stk)) \qquad (4.3\text{-}6)$$

is an implementation invariant. Again, this can be easily verified using data type induction. The invariant (4.3-6) is an example of a *representation invariant*, which we define to be that implementation invariant which describes how the abstract values are represented. The representation invariant can be constructed automatically from the *representation* part of the implementation.

As an example of an invariant of a data type other than an implementation invariant, consider

```
P(symtab) = (symtab = INIT)
          v (∃ symtab1, symtab = ENTERBLOCK(symtab1))
          v (∃ symtab1,id,attrlist,
               symtab = ADDID(symtab1,id,attrlist)).          (4.3-7)
```

which can be shown to satisfy each of the conditions in (4.3-4), using the syntactic specification and semantic axioms for the Symboltable data type. Invariants such as (4.3-7) are useful in proofs which make use of the data type, in that they can be used to reduce to a minimum the number of cases which must be considered in a proof by case analysis. Note that (4.3-7) can also be regarded as the representation invariant for the direct implementation of the Symboltable data type, as discussed in Section 2.3.

4.4 *An interactive data type verification system*

The programs which we have developed for use in verifying data type implementations are currently being run as part of an interactive verification system [Good75]. The complete implementation of the symbol table operations by a stack of hash tables has been verified using these programs. As the user-interface for the proof process is still under development, we will not give a detailed description of the commands of the current facilities (called the Data Type Verification System), but merely indicate the nature of the commands and overall proof process with an example, the verification of the top level implementation by a Stack of Mappings. The first step is to direct the system to adopt the programs of data type Symboltable, and the axioms of data types Stack and Mapping. These programs and axioms would all be in the form of rewrite rules which the user had just entered or had read in from files. The command for "adopting" a set of rules is separated from the act of reading them in so that several sets of rules for an operator can coexist within the system. Assuming the Symboltable axioms have also been input to the system, the user then directs the system to generate the verification conditions for the data type. These would consist of the Symboltable axioms and the equality axioms for the Symboltable equality operator (see [Guttag76c]), all interpreted in terms of the representation.

The user can then attempt to prove each of the verification conditions using CEVAL, which is an evaluator implementing the logical deduction rules discussed in Section 4.1. In these proofs the rewrite rules from the Symboltable programs and Stack and Mapping axioms are used automatically, without further direction from the user. In some cases, as noted in Section 4.3, completion of a proof will require one or more assumptions to be made about the representation or the Stack or Mapping data types. If this is the case the system will stop with a reduced form of the original verification condition. Examination of this output will often lead the user to the necessary assumptions. Initially these assumptions are input by the user and used as needed without justification. To complete the verification of the implementation, it is necessary to prove these assumptions, or a stronger set of assumptions, as invariants (of the Symboltable data type implementation or of the Stack or Mapping data types). The verification conditions sufficient to establish these invariants are constructed using the syntactic specifications of the data types, in accordance with the principle of data type induction, as described in Section 4.3.

5. *Conclusions*

Elsewhere it has been argued that abstract data types can be effectively employed as a "thought tool" in the structured development of programs ([Guttag 76b], [Standish 73], [Wulf 76], [Liskov 75]). In this paper we have attempted to show that the use of algebraic axioms as a means for describing data abstractions is also valuable, when properly used, for both formal and informal program validation.

In sections 2 and 3 we discussed the axiom language and some techniques for constructing algebraic axiomatizations, based on an important duality between programs and axioms. Parts of an implementation of a moderately complex symbol table were exhibited to show how these techniques might be used in practice. In Section 4 we demonstrated how properly written axioms can be used in formal program verification. Abstract data types provide a mechanism for factoring proofs into manageable sections. At one level of abstraction the axiomatic specification provides us with theorems that may be applied in the verification of programs that use abstract data types. We have illustrated how such axioms may be used to verify the correctness of implementations of higher-level abstract data types in terms of lower-level ones. Writing the axioms in a certain style allows them to be used as rewrite rules so that the proofs can in large part be carried out as a sequence of reductions. The feasibility of automating such proofs has been demonstrated with a prototype interactive verification system for data types.

It is important to notice that the techniques developed in this paper are essentially programming language independent. While the availability of languages with the compile time type facilities of SIMULA 67 [Dahl 68], CLU [Liskov 74], or Alphard [Wulf 76] will make these techniques easier to apply, they are by no means essential. If one exercises enough self discipline to ensure the validity of data type induction, the techniques described should prove useful in the development of programs in any language. The ultimate test of any research designed to enhance the process of software development must lie in the application of the results of that research to a large software project. We feel that the work described in this paper lays the groundwork necessary for such a test.

Acknowledgements

We would like to thank Dick Jenks, Ralph London, Nancy Lynch, Mark Moriconi, Jernej Polajnar, Dave Wile, and Marty Yonke for their careful reading of earlier drafts of this paper, and Betty Randall for her help in typing the manuscript.

References

[Boyer75] R. S. Boyer and J S. Moore, "Proving theorems about LISP functions," *J. ACM*, 22, 1, January 1975, 129-144.

[Dahl68] O.-J. Dahl, *The SIMULA 67 common base language*, Norwegian Computing Center, Oslo, 1968.

[Goguen75] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright, "Abstract data-types as initial algebras and correctness of data representations, *Proceedings*, Conference on Computer Graphics, Pattern Recognition and Data Structure, May 1975.

[Good75] D. I. Good, R. L. London, and W. W. Bledsoe, "An interactive program verification system," *IEEE Transactions on Software Engineering*, SE-1, 1, March 1975, 56-67.

[Guttag75] J. V. Guttag, "The specification and application to programming of abstract data types," Ph. D. Thesis, University of Toronto, Department of Computer Science, 1975, available as Computer System Research Report CSRG-59.

[Guttag76a] J. V. Guttag, "Abstract data types and the development of data structures," Supplement to the Proceedings of the SIGPLAN/SIGMOD Conference on Data: Abstraction, Definition, and Structure, March 1976, pp. 37-46.

[Guttag76b] J. V. Guttag, E. Horowitz, D. Musser, "The design of data type specifications," USC Information Sciences Institute Research Report, Marina del Rey, Ca., 1976 (to appear in Proc. Second International Conference on Software Engineering , San Francisco, October 1976).

[Guttag76c] J. V. Guttag, E. Horowitz, and D. R. Musser, "Abstract data types and software validation," USC information Sciences Institute Research Report, Marina del Rel, Ca., 1976.

[Hoare72] C. A. R. Hoare, "Proof of correctness of data representations,"*Acta Informatica*, 4, 1972, pp. 271-281.

[Horowitz76] E. Horowitz and S. Sahni, *Fundamentals of Data Structures*, Computer Science Press, June 1976.

[Liskov74] B. H. Liskov and S. N. Zilles, "Programming with abstract data types," *Proceedings of ACM SIGPLAN Symposium on Very High Level Languages, SIGPLAN Notices*, 9,4, April 1974, 50-59.

[Liskov75] B. H. Liskov and S. N. Zilles, "Specification techniques for data abstractions," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 1, March 1975, pp. 7-18.

[Manna74] Z. Manna, *Mathematical Theory of Computation*, McGraw-Hill, 1974.

[McCarthy63] J. McCarthy, "Basis for a mathematical theory of computation," in *Computer Programming and Formal Systems*, P. Braffort and D. Hirchberg (eds.), North-Holland Publishing Company, 1963, pp. 33-70.

[Parnas72] D. L. Parnas, "Information distribution aspects of design methodology," Proc. IFIP Congress 71, Vol. 1 (1972), pp. 339-344.

[Rosen73] B. Rosen, "Tree Manipulating systems and Church-Rosser theorems," *J. ACM*, 20,1, January 1973, 160-187.

[Spitzen75] J. Spitzen and B. Wegbreit, "The verification and synthesis of data structures," *Acta Informatica* 4, 127-144 (1975).

[Standish73] T. A. Standish, "Data structures: an axiomatic approach," BBN Report No. 2639, Bolt Beranek and Newmann, Cambridge, Mass., 1973.

[Wegbreit76] B. Wegbreit and J. Spitzen, "Proving properties of complex data structures," *J. ACM*, 23, 2, April 1976, 389-396.

[Wulf76] W. A. Wulf, R. L. London, and M. Shaw, "Abstraction and verification in Alphard: introduction to language and methodology," *Carnegie-Mellon University* and *USC Information Sciences Institute Technical Reports*, 1976.

[Zilles75] S. N. Zilles, "Abstract specifications for data types," IBM Research Laboratory, San Jose, California, 1975.

AN ON-LINE SESSION WITH THE STANFORD VERIFIER

by
David Luckham

The following is a transcript with comments of an on-line session with the Stanford verifier. A program was suggested at the beginning of the session by a visitor to the project. His program was simpler than most examples currently being experimented with using this verifier, but being a surprise, it enabled him to see some of what goes on in the process of verifying a new program. The session includes examples of both verification and debugging using this kind of system.

Comments are in lower case. commands typed by the user are prefixed by "#". Responses from the verifier are prefixed or headed by *'s.

We started with the program below with ENTRY/EXIT specifications stating that its output array is a permutation of its input; an INVARIANT assertion was included for each of the WHILE loops. PERMUTATION(A,B,L,R) means that array A is a permutation of B within the range of indices L to R.

```
PASCAL
TYPE NARRAY=ARRAY[1..N] OF INTEGER;

PROCEDURE SORT (VAR A:NARRAY);
INITIAL A=A0;
EXIT PERMUTATION(A,A0,1,N);
VAR TEMP:INTEGER;
VAR I,J:INTEGER;
BEGIN
I:=1; J:=1;
INVARIANT PERMUTATION(A,A0,1,N)
WHILE (I<N) DO
  BEGIN
     INVARIANT PERMUTATION(A,A0,1,N)
     WHILE (J≤N-I) DO
      BEGIN
      IF A[J]>A[J+1] THEN BEGIN TEMP:=A[J]; A[J]:=A[J+1]; A[J+1]:=TEMP  END;
      J:=J+1
      END;
     I:=I+1;
 END
```

The program was typed into a disk file called EX.

```
#R VERIFY 120                          call the verifier with 60k core.

#(PARSE EX)                            verify the program in file EX.
```

The first two results were syntax error messages due [1..N] instead of [1:N] (not standard Pascal), and a missing END. We omit these messages.

The corrected program was:

```
PASCAL
TYPE NARRAY=ARRAY[1:N] OF INTEGER;

PROCEDURE SORT (VAR A:NARRAY);
INITIAL A=A0;
EXIT PERMUTATION(A,A0,1,N);
VAR TEMP:INTEGER;
VAR I,J:INTEGER;
BEGIN
I:=1; J:=1;
INVARIANT PERMUTATION(A,A0,1,N)
WHILE (I<N) DO
  BEGIN
      INVARIANT PERMUTATION(A,A0,1,N)
      WHILE (J≤N-I) DO
       BEGIN
       IF A[J]>A[J+1] THEN BEGIN TEMP:=A[J]; A[J]:=A[J+1]; A[J+1]:=TEMP  END;
       J:=J+1
       END;
      I:=I+1;
      J:=1
  END
END;.;
```

comment: The following notation is used to express the result of operations on arrays: <A,[I],E> represents the array obtained from A by placing E in the Ith position.

#(PARSE EX)

The resulting four verification conditions were:

****

VERIFICATION CONDITION: SORT 1

****

```
(TRUE                                    corresponds to the path
→                                        from ENTRY to outer WHILE
 PERMUTATION(A,A,1,N) &                  to EXIT.
 (¬100<N &
  PERMUTATION(A00,A,1,N)
 →
  PERMUTATION(A00,A,1,N)))
```

****

VERIFICATION CONDITION: SORT 2

****

```
(I<N &
 PERMUTATION(A,A0,1,N)                   corresponds to the path
→                                        round the outer loop to
 PERMUTATION(A,A0,1,N) &                 inner WHILE to EXIT.
 (¬J02≤N-I &
  PERMUTATION(A02,A0,1,N)
 →
  PERMUTATION(A02,A0,1,N)))
```

****

VERIFICATION CONDITION: SORT 3

****

```
(¬A[J+1]<A[J] &
 J≤N-I &                                 corresponds to the path
 PERMUTATION(A,A0,1,N)                   round the inner loop
→                                        when the IF test is false.
```

```
 PERMUTATION(A,A0,1,N))

*****************************************

VERIFICATION CONDITION: SORT 4

*****************************************


(A[J+1]<A[J] &                          corresponds to path
 J≤N-I &                                round inner loop when
 PERMUTATION(A,A0,1,N)                  the IF test is true.
→
 PERMUTATION(<<A,[J],A[J+1]>,[(J+1)],A[J]>,A0,1,N))


*****************************************

SIMPLIFIED VERIFICATION CONDITION: SORT 1

*****************************************


(TRUE
→
 PERMUTATION(A,A,1,N))


*****************************************

SIMPLIFIED VERIFICATION CONDITION: SORT 2

*****************************************

TRUE


*****************************************

SIMPLIFIED VERIFICATION CONDITION: SORT 3

*****************************************

TRUE
```

****

SIMPLIFIED VERIFICATION CONDITION: SORT 4

****

```
(A[J+1]<A[J] &
 PERMUTATION(A,A0,1,N) &
 J+1≤N
→
 PERMUTATION(<<A,[J],A[J+1]>,[(J+1)],A[J]>,A0,1,N))
```

*****

TIME: 0 CPU SECS, 1 REAL SECS

comment: The simplification has been made without any non logical knowledge (e.g. the verifier has not been given any facts about the concept PERMUTATION(A,B,1,N). The simplified VC SORT 1 shows that it doesn't even know PERMUTATION(A,A,1,N).

Notice the notation for the array manipulation in the conclusion of VC SORT 4. In more complicated cases it can get pretty unreadable. Is a more readable notation possible (or needed)?

Our visitor now asked if we had a file of lemmas describing standard concepts used in verifying sorting programs. So we now read in a file of standard lemmas used for versions of QUICKSORT. The file contained lemmas about many concepts other than PERMUTATION.

#(PARSE (QUICK.GO)) read in the lemmas.

******

GOALFILE ADDED 2 CPU SECONDS

The set of lemmas added is below (the relevant parts are those dealing with PERMUTATION and EXCHANGE). EXCHANGE defines a shorthand for the kind of array transformation expression that appeared above in VC SORT 4.

```
GOALFILE        % Goals and axioms to verify quicksort. Also needs ARITH.GO %

% ISSORTEDARRAY(A,B,L,R) asserts that array A is a sorted form of array B in the
      interval [L,R] %

      AXIOM ISSORTEDARRAYOF(@A,@B,@L,@R) ↔ PERMUTATION(A,B,L,R) ∧
            ORDERED(A,L,R);
            % A is B sorted on [L,R] ↔ it is a permutation of B and is ordered %
% PERMUTATION(A,B,L,R) asserts that array A is a permutation of array B in the
      interval [L,R] %

      AXIOM PERMUTATION(@A,@A,@L,@R) ↔ TRUE;
            % An array is a permutation of itself %
      AXIOM IF (L≤P1) ∧ (P1≤R) ∧ (L≤P2) ∧ (P2≤R) THEN
            PERMUTATION(EXCHANGE(@A,@P1,@P2),@A,@L,@R) ↔ TRUE;
            % Exchanging two elements is a valid permutation %
      GOAL PERMUTATION(@A,@B,@L,@R) SUB PERMUTATION(A,@C,L,R) ∧
            PERMUTATION(@C,B,L,R);
            % Transitivity of the permutation relation %
      GOAL PERMUTATION(@A,@B,@L,@R) SUB PERMUTATION(A,B,@L1,@R1) ∧
            (L≤@L1) ∧ (@R1≤R) ∧ PRESERVEEXCEPT(A,B,@L1,@R1);
            % Permuting a subarray of A between L1 and R1 and leaving the rest
            unchanged is a valid permutation %
% EXCHANGE(A,P1,P2) is array A with the elements in positions P1 and P2 exchanged %

      AXIOM IF Y=B[P2] THEN
            <<@B,[@P1],@Y>,[@P2],@X> ↔ EXCHANGE(<B,[P1],X>,P1,P2);
% ORDERED(A,L,R) asserts that array A is in ascending order on the interval [L,R] %

      AXIOM IF R≤L THEN ORDERED(@A,@L,@R) ↔ TRUE;
            % An empty interval is ordered %
      GOAL ORDERED(@A,@L,@R) SUB ORDERED(A,L,@P-1) ∧ ORDERED(A,@P+1,R) ∧
            BIGGEST(A[@P],A,L,@P-1) ∧ SMALLEST(A[@P],A,@P+1,R) ∧ (L≤@P) ∧
            (@P≤R);
            % Array A is ordered on [L,R] if A[P] partitions the elements of A
            with those bigger in the interval [P+1,R] and those smaller in the
            interval [L,P-1], and each subarray is ordered %
      GOAL ORDERED(@A,@L,@R) SUB ORDERED(@B,L,R) ∧ PRESERVEEXCEPT(A,@B,@L1,@R1)
            ∧ ((@R1<L) ∨ (R<@L1));
            % Array A remains ordered on [L,R] if it was ordered and nothing in
            the interval [L,R] has been changed %

% BIGGEST(X,A,L,R) asserts that X is ≥ anything in array A in the interval [L,R] %

      AXIOM IF L>R THEN BIGGEST(@X,@A,@L,@R) ↔ TRUE;
            % X is bigger than anything in an empty interval %
      GOAL BIGGEST(@X,@A,@L,@R) SUB BIGGEST(X,A,L,R-1) ∧ (A[R]≤X);
```

```
        % X is BIGGEST on [L,R] if it is BIGGEST on [L,R-1] and ≥ A[R] %
    GOAL BIGGEST(@A[@M1],@A,@L,@R) SUB BIGGEST(@B[M1],@B,L,R) ∧
        ((@R1<L) ∨ (R<@L1)) ∧ ((@R1<M1) ∨ (M1<@L1)) ∧
        PRESERVEEXCEPT(A,@B,@L1,@R1);
        % A[M1] remains BIGGEST on [L,R] if no changes have been made to
        either A[M1] or any A[I] in the interval [L,R] %
    GOAL BIGGEST(@X,<@A,[@R1],@Y>,@L,@R) SUB BIGGEST(X,A,L,R) ∧ (R1>R);
        % X remains BIGGEST on [L,R] if the change to A occurs to the right
        of the interval %

    GOAL BIGGEST(@A01[@X],@A01,@L,@R)
     SUB PRESERVEEXCEPT(A01,@A,L,R) ∧
        BIGGEST(A[X],A,L,R) ∧
        PERMUTATION(A01,A,L,R)∧
        (X>R);
        %If A[X] is the biggest elt. of A in the range [L,R],and X is outside
        the range, and A01 results by permuting this range, then A01[X] will
         remain the biggest elt of A01 in the range [L,R]%

% SMALLEST(X,A,L,R) asserts that X is ≤ anything in array A in the interval [L,R] %

    AXIOM IF L>R THEN SMALLEST(@X,@A,@L,@R) ↔ TRUE;
        % X is smaller than anything in an empty interval %
    GOAL SMALLEST(@X,@A,@L,@R) SUB SMALLEST(X,A,L+1,R) ∧ (X≤A[L]);
        % X is SMALLEST on [L,R] if it is SMALLEST on [L+1,R] and X≤A[L] %
    GOAL SMALLEST(@A[@M1],@A,@L,@R) SUB SMALLEST(@B[M1],@B,L,R) ∧
        ((@R1<L) ∨ (R<@L1)) ∧ ((@R1<M1) ∨ (M1<@L1)) ∧
        PRESERVEEXCEPT(A,@B,@L1,@R1);
        % A[M1] is SMALLEST on [L,R] if it was before and neither A[M1] nor
        anything in [L,R] has been changed %
    GOAL SMALLEST(@X,@A,@L,@R) SUB SMALLEST(X,A,@L1,R) ∧ (@L1≤L);
        % SMALLEST of a subfile is true %
    GOAL SMALLEST(@X,<@A,[@L1],@Y>,@L,@R) SUB SMALLEST(X,A,@L2,R)∧(L1<L)∧(@L2≤L);
        % X remains smallest on [L,R] if it was on a larger interval and the
        change to A occurred to the left of L %

    GOAL  SMALLEST(@A00[@X],@A00,@L,@R)
     SUB  SMALLEST(@A[X],@A,L,R) ∧
        PERMUTATION(A00,A,L,R) ∧
        PRESERVEEXCEPT(A00,A,L,R) ∧
        (X<L);
        %A00[X] remains the smallest elt. in the range [L,R] when
         A00 is obtained from A by permuting [L,R] and X is outside
         this range.%


% PRESERVEEXCEPT(A,B,L,R) asserts that any differences between arrays A and B occur
```

```
        IN the interval [L,R] %

        AXIOM PRESERVEEXCEPT(@A,@A,@L,@R) ↔ TRUE;
              % If nothing changes, everything is preserved %
        AXIOM IF (L≤P) ∧ (P≤R) THEN PRESERVEEXCEPT(<@A,[@P],@X>,@A,@L,@R) ↔ TRUE;
              % Changes within [L,R] preserve things outside of [L,R] %
        GOAL PRESERVEEXCEPT(EXCHANGE(@A,@R1,@L1),@B,@L,@R) SUB PRESERVEEXCEPT(A,B,L,
              R) ∧ (L≤L1) ∧ (L≤R1) ∧ (L1≤R) ∧ (R1≤R);
              % Exhanges within [L,R] preserve things outside of [L,R] %
        GOAL PRESERVEEXCEPT(@A,@B,@L,@R) SUB PRESERVEEXCEPT(A,B,@L1,@R1) ∧
              (L≤@L1) ∧ (@R1≤R);
              % Nothing changes outside of [L,R] if nothing changes outside of a
              subinterval of [L,R] %
        GOAL PRESERVEEXCEPT(@A,@B,@L,@R) SUB PRESERVEEXCEPT(A,@C,L,R) ∧
              PRESERVEEXCEPT(@C,B,L,R);
              % Transitivity of PRESERVEEXCEPT %

% These are some inequality rules that the verifier could not deduce from ARITH.GO %

        GOAL @X≤@X SUB TRUE;
        GOAL @X≥@X SUB TRUE;
        GOAL @X>@X-1 SUB TRUE;
        GOAL @X<@X+1 SUB TRUE;
        GOAL @X-1≤@Z SUB (@X≤@Y) ∧ (@Y≤@Z) ;
        GOAL @X≤@Y+1 SUB X≤Y;
        GOAL @X≤@Y SUB X<Y;

% Tell it what an ∨ is%

        GOAL @A ∨ @B SUB A,B;
        .;
```

#(SIMPLIFY SORT 1) Now try to prove VC SORT 1.

```
**********************************************

SIMPLIFIED VERIFICATION CONDITION: SORT  1

**********************************************

TRUE
```

#(SIMPLIFY SORT 4)

```
**********************************************

SIMPLIFIED VERIFICATION CONDITION: SORT  4

**********************************************

(A[J+1]<A[J] &
 J+1≤N
→
 PERMUTATION(EXCHANGE(A,J,J+1),A,1,N))

*****

TIME: 0 CPU SECS, 0 REAL SECS
```

comment: Notice that the array manipulation expression has been replaced by the convenient EXCHANGE notation. The replacement rules permit the user to define his own notation.

VC SORT 4 was not proved because the relevant lemma requires facts about the range bounds (L and R in the second AXIOM in the PERMUTATION lemmas). In the simple program here these bounds are fixed at 1 and N, so a special case of this lemma will suffice.

We now add a simple extra lemma on-line:

```
#(PARSE)
#GOALFILE
#R1:INFER PERMUTATION(EXCHANGE(@A,@J,@J+1),@A,1,N));;

******
GOALFILE ADDED 0 CPU SECONDS
```

The rule typed in on-line has been named R1.

```
#(SIMPLIFY SORT 4)

*****************************************

SIMPLIFIED VERIFICATION CONDITION: SORT  4

*****************************************

USING RULES  R1,

TRUE
```

Next we added a bug suggested by the visitor (the bug is in the line marked by #):

```
PASCAL
TYPE NARRAY=ARRAY[1:N] OF INTEGER;

PROCEDURE SORT (VAR A:NARRAY);
INITIAL A=A0;
EXIT PERMUTATION(A,A0,1,N);
VAR TEMP:INTEGER;
VAR I,J:INTEGER;
BEGIN
I:=1; J:=1;
INVARIANT PERMUTATION(A,A0,1,N)
WHILE (I<N) DO
  BEGIN
     INVARIANT PERMUTATION(A,A0,1,N)
     WHILE (J≤N-I) DO
      BEGIN
#      IF A[J]>A[J+1] THEN BEGIN TEMP:=A[J]; A[J]:=A[J+1]; A[J]:=TEMP  END;
      J:=J+1
      END;
     I:=I+1;
     J:=1
  END
END;.;
```

#(PARSE EX) where EX now contains the buggy program led to the result:

```
*****************************************

SIMPLIFIED VERIFICATION CONDITION: SORT  1

*****************************************

TRUE


*****************************************

SIMPLIFIED VERIFICATION CONDITION: SORT  2

*****************************************

TRUE


*****************************************

SIMPLIFIED VERIFICATION CONDITION: SORT  3

*****************************************

TRUE


*****************************************

SIMPLIFIED VERIFICATION CONDITION: SORT  4

*****************************************

TRUE

*****

TIME: 1 CPU SECS, 1 REAL SECS
```

comment: very interesting! This kind of thing always happens in demonstrations. However, if you look back at the buggy program you will see that the "bug" does not invalidate the PERMUTATION property; it affects the ORDEREDness.

So we tried another bug. The result was this:

```
**********************************************

SIMPLIFIED VERIFICATION CONDITION: SORT  1

**********************************************

TRUE


**********************************************

SIMPLIFIED VERIFICATION CONDITION: SORT  2

**********************************************

TRUE


**********************************************

SIMPLIFIED VERIFICATION CONDITION: SORT  3

**********************************************

TRUE
```

****

SIMPLIFIED VERIFICATION CONDITION: SORT 4

****

(A[J+1]<A[J] &
J+1≤N
→
PERMUTATION(<A,[J],A[J+1]>,A,1,N))

*****

TIME: 1 CPU SECS, 2 REAL SECS

comment: The only change made has been to the code. The first three VC's are proved. By looking at VC SORT 4 can you tell what the bug is, and where it is? You can figure out the error generated by the bug from the array expression <A,[J],A[J+1]>. The boolean tests determining the path on which the bug occurs are in the premisses of this VC.

Here's the second buggy program.

```
PASCAL
TYPE NARRAY=ARRAY[1:N] OF INTEGER;

PROCEDURE SORT (VAR A:NARRAY);
INITIAL A=A0;
EXIT PERMUTATION(A,A0,1,N);
VAR TEMP:INTEGER;
VAR I,J:INTEGER;
BEGIN
I:=1; J:=1;
INVARIANT PERMUTATION(A,A0,1,N)
WHILE (I<N) DO
  BEGIN
     INVARIANT PERMUTATION(A,A0,1,N)
     WHILE (J≤N-I) DO
      BEGIN
      IF A[J]>A[J+1] THEN BEGIN TEMP:=A[J]; A[J]:=A[J+1]  END;
      J:=J+1
      END;
     I:=I+1;
     J:=1
  END
END;.;
```

The whole session including my one finger typing of the program etc. took about twenty minutes of console time.

We are pursuing three areas of experimental research with this verifier at present. (1) The verifier is currently being developed to exploit its use as a debugging aid. (2) We are experimenting with classes of more complex programs (including pointer manipulations) and with methods of verifying larger programs than have been done so far. (3) The standardization of verifications (i.e. the required lemmas) so that changes in code can be checked quickly using a verification of a previous version of the program is being attempted for several kinds of program. This should lead to applications to the software maintainence problem. Further information on this project can be found in the Stanford Artificial Intelligence Laboratory reports, Stanford University (or by contacting the author).

ACKNOWLEDGEMENTS The goalfile is taken from Scott Drysdale's work on debugging and verifying six versions of QUICKSORT. The visitor was Bill Carlson of ARPA.

Gypsy: A Language for Specification and Implementation of Verifiable Programs

Allen L. Ambler, Donald I. Good,
James C. Browne, Wilhelm F. Burger,
Richard M. Cohen, Charles G. Hoch,
and Robert E. Wells
The University of Texas at Austin
Austin, Texas 78712

Introduction

Gypsy is the unifying element of a complete methodology for the construction of rigorously verified systems programs. This methodology provides an integrated way of specifying, designing, and implementing programs and of verifying that they always execute in conformity with their stated specifications, even in imperfect execution environments. The Gypsy language provides a precise means of expressing both a program and its specifications from initial specification through program design, implementation, verification, and successive modification and evolution. This integration of programming and specification facilities into a common language is the most significant single characteristic of Gypsy. This requires the program designer and verifier to comprehend only a single syntax and semantics throughout program development. This also allows program proofs to be constructed rigorously throughout all stages of development, thereby bringing maximal benefit to the total programming process.

The incorporation of specifications and programming facilities into a single language provides three complementary approaches to program verification. First, formal proofs that the program meets specifications can be constructed before any execution of the program occurs. Second, specifications can be validated by actual evaluation at run-time. Third, trace facilities provide a convenient mechanism for post-execution analysis if desired. This establishes a very effective relationship between program proof and run-time validation of specifications. Those specifications that are validated at run-time need not be proved, but can be assumed in the proofs, thus reducing significantly the size and complexity of the formal proofs. This, of course, increases program execution time, but provides the program designer and verifier with a choice of implementation and verification strategy.

One of the common, and often unstated, assumptions of formal program proofs is that of a perfect execution environment. For example, the problems of arithmetic on a finite set of integers often are ignored. Also it invariably is assumed that if an assignment x:=a is executed, then a successive reference to the value of x will equal a. In reality, this normally would imply that a computer memory never will drop a bit in the word(s) where x is stored. In Gypsy the conceptual span of both specifications and program code has been extended to include program execution in imperfect environments. Specifications and program code concerning data integrity, error monitoring, and error isolation and recovery are expressed in a unified form along with the error-free environment statements.

In theory, methods of program proof can be applied to programs of any size. In practice, however, the provability of a program depends directly on the degree to which a program can be decomposed into small, independently provable units. Gypsy supports this kind of decomposition through facilities for both operational and data abstraction. The data abstraction is provided through a general mechanism for access control. These abstraction facilities can be applied uniformly either to programs or to specifications, and provide an effective basis for incremental design and verification of a complete program. Gypsy further supports this process by providing explicit facilities for top-down development. This permits higher-level units to be designed and verified even though some of the lower level details of their implementations may still be pending.

The original target of Gypsy was the expression of verifiable programs for communications processing such as those that might be found at the node of a computer network. This led to the incorporation of verifiable features for expressing concurrency and process synchronization and for expressing real-time dependencies. The result has been a high-level language for the development of general systems programs that can be verified to execute in conformity with precisely stated specifications.

Design of Gypsy

Gypsy was developed as an integrated programming and specification language to support the complete design, specification, coding and verification of systems software, with particular emphasis on communications software. Specific goals were:

* Complete Verifiability. Every feature in the language must be rigorously verifiable.

* Incremental Development. The language must support modular, incremental program development and verification. As best possible the language must simplify the verification process by encouraging small modules with tightly regulated interactions and by isolating and minimizing the effects of modifications to previously verified code. There must also be a facility for partial expression of program units.

* Systems Programming. The language must support the development of systems software. There must be facilities for expressing process concurrency and synchronizing process communication. There must also be facilities for expressing real-time dependencies.

* Imperfect Execution Environments. The language must support execution in imperfect environments. It must be possible to detect, isolate, and recover from run-time anomalies as well as monitor the program state.

* Specification Capability. The language must provide an extensive specification capability. For every property that is to be verified, there must be an adequate

means of expressing it directly in the language. The integration of formal proof, run-time validation, and monitoring must be consistent and provide a complete whole.

The design of Gypsy has been a combined process of synthesis and contraction. Starting from Pascal [12] each existing Pascal construct was carefully analyzed and those which inhibited verification were modified or removed. The hierarchical definition structure was eliminated and protection lists were added to provide a tighter, more flexible environment for incremental program development and verification. Facilities for expressing concurrency, communication, synchronization, timing constraints, external events, error recovery, and monitoring were added, paying close attention to the requirements of the verification methodology. Each construct in the program code and the specification statements was designed to support the verification methodology. The program code syntax was modified to integrate these specification statements into a logically consistent and hopefully understandable language.

Designing for Verification

A language which is to facilitate coding and specification must not only include capabilities necessary for expressing the problem domain of interest, but must exclude language constructs whose semantics defeat, or impede, verification. We defer a discussion of Gypsy's specification statements until a later section for pedagogical reasons. Their development was, however, closely interwoven with that of the coding statements.

Verification of program code has only recently become a prominent factor in programming language design. While Pascal was influenced by verification considerations [5], more recently Nucleus [10] and Alphard [21] have been expressly designed for verification by formal proofs. Gypsy also is specifically designed for verification, but verification by run-time validation as well as by formal proof. The first phase in the design of Gypsy was to develop a "conventional" language which was free from concepts known to render formal proof verification difficult. To this end, Pascal [12] was selected as a model and Gypsy was patterned after Pascal, but with significant differences.

Routines in Pascal can be nested to arbitrary depths which creates a hierarchy of nested "non-local" variables. Routines in Gypsy may not be nested and variables can only be defined within routines; hence, Gypsy has no non-local variables. This simplifies verification as well as incremental program development, which will be discussed in the next section.

Functions in Pascal can take either variable or value parameters and can only return values of a simple type. In Gypsy, functions are allowed constant and value parameters and they can return values of any type. The restriction to these kinds of parameters, together with the absence of non-local variables, guarantees that functions produce no side-effects. This simplifies verification considerably.

Pascal allows routines to be included as parameters to other routines; Gypsy does not. This decision was made not because of the necessity to do dynamic type checking, but because the extra burden on the verification process did not appear worth the extra capability.

Certain of Pascal's data types do not appear at all in Gypsy. These are types "real", "class", "pointer", and "file".

Pascal has "if", "case", "for", "while", "repeat", and "goto" statements for execution control. Gypsy has a similar set of statements, "if", "case", "loop", and "leave", modified for proper placement of assertions and to eliminate the need for bracketing "begin-end" pairs. The "if" statement is conventional except for a trailing "end". The "case" statement has an additional keyword "is" and an optional "else" clause. The "loop" statement subsumes both the "while" and "repeat" constructs as well as the so-called "loop-and-a-half" construct and infinite loops. Termination and looping are controlled by "leave" statements. Gypsy has no "goto" statement.

Designing for Incremental Development

A language that is to support the development and evolution of verified programs also must consider the practical aspects of verification. In developing a verified program of any significant size, it is necessary that the program be written as a large collection of small, independently verifiable units. Otherwise, a formal proof easily can expand into a mass of detail and become unmanageable. Also for proofs to be maximally effective they should be carried out on a unit-by-unit basis as the program is developed. Further, it is the nature of systems programs that they are continuously undergoing evolution and with each modification some amount of reverification is necessary. It is, therefore essential that the amount of reverification be kept to a minimum. For these reasons, we sought language features which supported unit-by-unit manipulation, increased unit independence, and isolated unit interactions.

A Gypsy program consists of a series of "routine", "macro", "constant", and "type" units; which may appear in any order. If a reference can not be resolved locally within a particular unit, a search of the other external unit names is made. When an unresolved local reference is found to be an external unit name, then the appropriate information is extracted and the analysis continued. Access rights to any unit may be stated in an "access list." These access lists will be checked during the process of resolving references. The combination of units and access lists provides a high degree of code independence, plus a tightly controlled environment.

A routine is a "function", a "procedure", a "process", or a "program". A "program" unit defines an initial entry point. Routine declarations can only appear at the unit level; hence, Gypsy does not provide a nested hierarchy. Besides favoring unit independence, it was felt (1) that a hierarchical structure failed to provide adequate program protection without access lists and (2) that with access lists and without nonlocal variables a hierarchical structure was unnecessary.

A macro unit binds a parameterized expression to a name. While macro expansions can be nested, they may not be recursively expanded as there would be no way to terminate a recursive expansion.

A constant unit parallels Pascal's constant declarations except that a constant may be of any type including a structured type. This provides the means for referencing global values without allowing global variables or requiring them to be passed as parameters if they are not to be modified.

A type unit declares a new type either by itemizing its value set or by composing existing types. A type unit which includes an access list is the equivalent of an abstract data structure [7] [13] [21] [3] [9]. The intent of an abstract data type is to be able to construct a new type and to restrict access to the components of that type to operations representative of the type. It should be possible with a proper implementation to alter the implementation of the abstract type and the corresponding operations without impacting the program which employs the abstract type.

Designing for Concurrency and Real-Time

Programming languages have traditionally avoided concurrency; there have, however, been exceptions. The Burroughs family of extended Algol languages [14] provide processes and process communication, Bliss [19] provides coroutines and processes, Concurrent Pascal [3] combines processes and monitors, and Algol 68 [18] provides collateral elaboration of clauses. Several other languages have primitive means of accessing operating system functions which provide concurrency. Operating system research has generated a large number of concurrency and synchronization techniques which we will not attempt to reference. Two systems, RC4000 [1] and HYDRA [20], which were significant factors in our decision on how to specify and implement concurrency.

Gypsy has a routine type called a "process". It differs from a "procedure" only in the types of parameters allowed and in the manner of its invocation. Processes communicate only through message buffers [2]. A message "buffer" is a finite length queue on which there are only two operations defined, "send"(enqueue) and "receive"(dequeue). The queue is manipulated by a strict FCFS algorithm. Whenever a "send" is made on a full buffer the sending process is suspended until the condition is remedied. Likewise, a "receive" on an empty buffer will cause the process to be suspended. Associated with every buffer is a semaphore which guarantees mutually exclusive access to the buffer.

Concurrent processes are initiated by a "cobegin..end" statement and may or may not terminate. Only when all processes called within a "cobegin" statement terminate will the statement following the "cobegin" be executed.

Polling is an important function of real-time systems; hence, it must be possible to poll a buffer without being suspended indefinitely trying to receive from an empty buffer. Gypsy has an "await" statement which allows the simultaneous waiting on the completion of any one of several buffer operations. An "await" is in many respects a guarded command [8], except that it has a very restricted set of guards and it has an optional time-out clause. The time-out clause specifies what is to be done if none of the requested operations completes by a certain time.

The concept of (real) time is provided by "clock" variables. A clock variable is a special variable which may not be modified by the program, but which is always changing. There may be any number of clocks in a program, but there is no guarantee that they will be synchronized. Gypsy programs may be distributed across many machines and this makes synchronization virtually impossible.

Designing for Imperfect Execution Environments

An attribute of real-time software often overlooked in programming languages is the existence of both hardware and software faults. Fault detection, isolation, and recovery is an essential function in real-time software and consequently, languages for expressing such software should (1) provide capabilities for fault control programming and (2) provide an interface to the hardware which allows for the detection, isolation, and recovery of faults. The work of the Newcastle group [16] represents virtually all of the previous efforts on this topic.

A "condition" in Gypsy is an instantaneous event which may occur during the execution of a program. There is a large class of predefined "conditions" which correspond to hardware errors and dynamic language semantics errors, such as "caseerror". Programmers may, in addition, name and signal fault conditions by using a "signal" statement or an "otherwise" clause on a specification (discussed in the next section).

Any statement ending with the word "end", may optionally end with a "condition clause" followed by the word "end". The effect of the condition clause is that whenever a condition occurs, an immediate branch is taken to the condition clause, of the innermost containing statement, which specifies an action for that condition. Searching for the innermost condition clause may involve exiting a routine. After the condition clause is executed, control does not return to where it was before the fault, but instead drops out of the statement whose condition clause was executed. In some sense, a condition clause is a restricted version of a PL/I "on" condition which resembles one of Zahn's event driven case statements [22].

Designing for Specification

Gypsy plays the dual role of programming and specification language. The specification component of the language permits the precise expression of desired functional properties of key parts of the program. These properties are stated in terms of valid states that are to be maintained on the data objects of the program at various points in the program computation. The objective of a verification is to show that the computation always proceeds in conformity with the stated specifications. The conformity of the program with its specification can in most

cases be either proved prior to execution or validated during execution. The same specification methods are used in both approaches to verification.

All specifications in Gypsy are stated as boolean-valued expressions. These specifications are designated to be verified either by proof, by run-time validation, or simply assumed. Specifications that are proved or assumed need not be evaluated at run-time, and therefore, they are permitted to contain special operations and types that could not otherwise be permitted. For example, boolean expressions may contain the logical quantifiers "for all" and "there exists" and refer to rational numbers and infinite sequences.

The most familiar kinds of specifications used in Gypsy are the "entry", "exit", and "assert" statements for procedures and functions. These follow the same form as that introduced by [11] for proving Pascal programs. The "exit" specification is interpreted in the weak sense, i.e. it holds if the program terminates.

"Entry", "exit", and "assert" specifications also can be used with processes. However, processes often are intentionally programmed never to terminate, and therefore an "exit" specification may be of no value. Specifications can be stated for non-terminating processes through "block" specifications. A "block" specification holds whenever the process is suspended by a buffer operation. This provides a temporary halting point.

Specifications for routines performing buffer manipulations normally are stated in terms of effects on buffer histories. In the terminology of [6], these are "mythical variables", but they are provided in a uniform way rather than being installed by the programmer. Associated with every buffer b are several histories that are relevant to specifications and to the proof methodology. For example, "b.infrom" refers to the sequence of objects received in from the buffer by the process, and "b.outto" is the sequence of objects sent out to the buffer from the process.

Any sequence of "var" declarations can be followed by a "keep" specification. The "keep" expression must be maintained throughout the immediate scope of the "var" declaration. A procedure or function call releases the "keep", but the called unit must reestablish it before returning. This type of assertion is similar to those used by [17] for run-time validation.

Routines that have access to the internal structure of a type have both internal and external specifications. This follows the specification methods of Alphard forms [21]. External specifications are visible to the outside environment; internal ones are not. The external specifications are the "entry", "block", and "exit" specifications. "Centry", "cblock", and "cexit" are the corresponding internal specifications. The internal specifications may refer to the internal (concrete) structure of accessible types; the external specifications may not.

Two kinds of specifications can be stated for Gypsy type definitions, "require" and "axiom". The require specification follows Alphard and is a precondition on the type parameters that is necessary for the proper creation of an object of that type. The axiom is a relation among the functions that have access to the type.

This set of specification methods provides powerful mechanisms for stating functional properties of programs, and formal proof methods have been defined for proving each of these types of properties. The specifications do not, at this time, directly permit the definition of quantitative aspects of program behavior such as resource utilization.

A Message Switching Network

The following example follows part of the development of a simple message switching network and illustrates many of the important features of Gypsy. Only the specification and implementation of the network will be discussed. Its verification is beyond the scope of this paper. The development of the network will be top-down, but Gypsy admits any kind of program design strategy.

The top-level structure of the network is shown in figure 1. The network switches messages among a fixed number of users, each of which communicates with the network through a port. We will ignore protocols, and assume that each message is a separate, complete communication. Even at this early stage of development, the network can be written in Gypsy.

```
program Network(var upa:PortArray) = pending;

type PortArray = array(UserId) of Port;

type UserId = integer[1..NUsers];
const NUsers:integer = pending;

type Port = record(Get,Put:Line);
type Line = buffer(CSize) of Message;
const CSize:integer = pending;

type Message = pending;
```

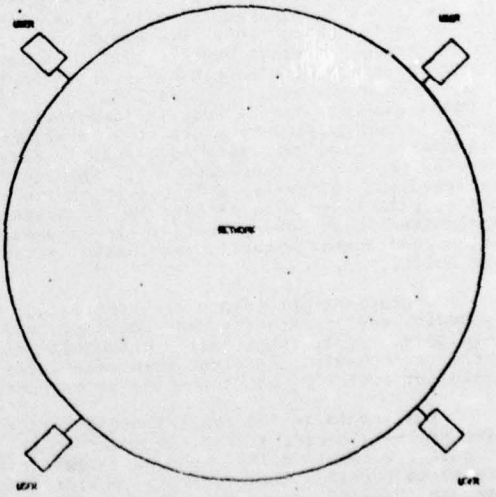


Figure 1

This program gives a precise description of the lines of communication between the network and its external environment. Communication is through an "array(UserId) of Port." Each port is a record consisting of two buffers, and each buffer contains a maximum of csize messages. The type userid is an integer restricted to the range [1..nusers]. The actual number of users, the maximal buffer size, the structure of messages, and the implementation of the network are left pending.

In a simple network with no protocols, the fundamental specification that is desired is that messages are delivered properly among all possible pairs of users. This specification for the network can be written as

```
program Network(var upa:PortArray) =
begin
  block all i,j:userid,
      ProperDelivery(i,j,upa);
  pending;
end;
```

The specification is written as a block, instead of an exit, specification because we intend the message network to be non-terminating. The network being blocked means that all processes in the network are blocked. This could happen for any number of reasons, including deadlock, but in this example, it will mean that there is no further input available from any user.

Before we can proceed with the implementation of the network, it is necessary that we be more specific about the meaning of "properdelivery." Loosely speaking, what we mean is that user j receives only those messages that were intended for j. We will make this definition precise with a macro.

```
define ProperDelivery(i,j,pa) =
  mail(pa(j).Put.outto,i,j)
      sub mail(pa(i).Get.infrom,i,j);
```

(The macro definition was chosen to illustrate the use of macros. ProperDelivery also could have been defined using a function.) Pa(j).put.outto is the sequence of all messages sent out to buffer pa(j).put by the network, and pa(i).get.infrom is the sequence of messages received in from buffer upa(i).get. The function mail(ms,i,j) is the subsequence of messages in message sequence ms that are directed from port i to j.

The completion of the definition of properdelivery requires a precise definition of the mail function, and mail in turn will require some additional information about messages.

```
function mail(ms:MessageSequence;i,j:UserId)
    :MessageSequence =
begin
  exit (assume mail(ms,i,j) =
    if ms=MessageSequence()
    then MessageSequence()
    else if i = Source(first(ms)) and
          j = Destination(first(ms))
      then MessageSequence(first(ms))
          @ Mail(nonfirst(ms),i,j)
      else Mail(nonfirst(ms),i,j)
    fi   fi);
end;

type MessageSequence = sequence of Message;

type Message<Source,Destination,Text,Compose,
    Equal> =
begin
  axiom all m:Message,
      Equal(Compose(Source(m),Destination(m),
          Text(m)),m);
  pending;
end;

function Source(m:Message):UserId =
    pending;
function Destination(m:Message):UserId =
    pending;
function Text(m:Message):CString =
    pending;
function Compose(s,d:Userid;t:CString)
    :Message = pending;
function Equal(m1,m2:Message):boolean =
    pending;

type CString = sequence (100) of char;
```

The definition of mail(ms,i,j) is given as an assumed exit specification which gives a complete recursive definition of mail. The definition of mail requires a new type, messagesequence. The type definition "sequence of message" defines a potentially infinite sequence of messages. Sequences are given a precise meaning by the semantics of Gypsy, but it is not necessary that they be implemented. Gypsy has a number of these kinds of constructs. They are included for purposes of formal program analysis, and may appear anywhere in a program where execution is not required, such as in specifications that are proved or assumed. In contrast, the type cstring is a sequence of ASCII characters of maximal size 100. Normally a Gypsy implementation would contain finite sequences but not infinite ones. Size restrictions can be enforced by run-time checks, and both kinds of sequences share a common semantics. Messagesequence() denotes the empty sequence of messages. In general, type names can be used to construct objects of that type. The @ operator is the sequence append operator.

The definition of mail makes use of two functions on messages, source and destination. The type definition of message permits these functions, as well as text, compose, and equal, access to the internal structure of messages, which is left pending. The axiom states an identity relation that must be maintained among this set of functions. This axiom implies that three kinds of information can be extracted from a message, a source, destination, and text part. The source and destination are the means of directing a message from one user to another, and the text is the actual content of the message to be transmitted. The compose function builds a message from these three parts, and equal defines a message equality. This is the only information that we will need to know about messages to carry out the full specification, implementation, and verification of the network process. Eventually, of course, we must choose a concrete representation of messages, and prove that the representation and the implementation of the functions that can access it satisfy the axioms.

Now we can give a completely precise interpretation to properdelivery. For every i,j pair, the mail from source i that is sent out to port j must be a subsequence of the mail received in from port i that is designated for destination j. This requires that the messages be the same and that they arrive in the same order that they were sent. The subsequence relation permits the network to drop messages. This is a concession to the reality of potentially unrecoverable transmission failures. This completes the specification of network.

We can proceed with the top-down design at any place in the current Gypsy program where a pending appears. There are many ways this program could be implemented to satisfy the block specification, but we will choose the following.

```
program network(var upa:PortArray) =
begin
  block all i,j:userid,
      ProperDelivery(i,j,upa);
  var npa:PortArray;
  cobegin
    Node(upa(i),npa(i),i) each i : UserId;
    switch(npa);
  end;
end;

process Node(var up,np:Port;i:UserId) =
begin
  block up.Put.outto sub np.Put.infrom
      and np.Get.outto sub up.Get.infrom;
  pending;
end;

process Switch(var npa:PortArray) =
begin
  block all i,j:UserId,
      ProperDelivery(i,j,npa);
  pending;
end;
```

This implements the program as a star network where each user is attached to exactly one node, and all of the nodes are connected to a single switch as shown in figure 2. Each node is similar to a full-duplex channel program passing messages unaltered, and in sequence, between the user and the central switch. All of the nodes and the switch are set into concurrent execution by the cobegin in the network program.

A node can be implemented by decomposing it into two one-way channels operating asychronously.

```
process Node(var up,np:Port;i:UserId) =
begin
  block up.Put.outto sub np.Put.infrom
    and np.Get.outto sub up.Get.infrom;
  cobegin
    Pass(np.Put,up.Put,i,Depart);
    Pass(up.Get,np.Get,i,Arrive);
  end;
end;

process Pass(var x,y:Line; i:UserId;
    d:Direction) =
begin
  block y.outto sub x.infrom;
  var m:Message;
  loop
    assert y.outto sub x.infrom;
    receive m from x;
    trace i = if d = Depart
        then Destination(m)
        else Source(m) fi;
    send m to y;
  end;
end;

type direction = (Arrive, Depart);
```

Pass is intentionally programmed as a non-termination loop. The loop simply receives messages from line x and passes them on to line y performing a trace depending on the value of d. The send and receive statements are potential blockage points, and these are the points where the block specification must hold. A cobegin, as in node or network, also is a potential blockage point.

The switch process also loops forever waiting on each buffer in its turn for a small time slice. If input is ready it will receive it; otherwise, it will time out and go on to the next buffer.

```
process Switch(var npa:PortArray) =
begin
  block all i,j:UserId,
      ProperDelivery(i,j,npa);
  var m:Message;
  var k:UserId;
  cond DestinationErr;
  keep Destination(m) in [1..NUsers]
      otherwise DestinationErr;
  loop
    k := 1;
    loop
      if k > NUsers then leave end;
      assert all i,j:UserId,
          ProperDelivery(i,j,npa);
      await
      on receive m from npa(k).Get:
          send m to npa(destination(m)).Put;
      after TimeSlice: ;
      when
      is DestinationErr: ;
      end;
      k := k + 1;
    end;
  end;
end;

const TimeSlice:integer = pending;
```

Switch repeatedly iterates through the get buffers of the ports attempting to receive a message. Control leaves the inner loop at the leave statement, and the outer loop runs indefinitely. If a message is not received in timeslice amount of time, the await is exited and the next buffer is considered. If a message is received within the allocated amount of time, it is sent to the appropriate destination. The keep specification of switch is evaluated each time one of its variables is assigned a new value. If the specification ever is violated, a

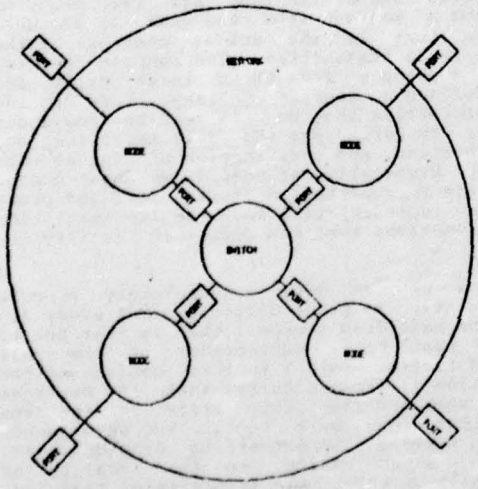


Figure 2

destination error is signalled. The keep prevents an invalid array index in the send statement. If the error occurs, control is transferred to the when part of the await and the destinationerr part of the when is performed. In this case, switch does nothing, thus dropping the message. This conforms with the subsequence relation specified in proper delivery.

The process structure of the complete network is shown in figure 3. All of these processes run concurrently. The intermediate level of a node process was not necessary. The pass processes could have been invoked explicitly from the cobegin in the network. The extra level of decomposition is helpful conceptually and in breaking the network into small, individually verifiable components.

Now let us return to the implementation of messages. They will be implemented in the obvious way as a record of three fields.

```
type Message<Source,Destination,Text,Compose,
    Equal> =
begin
  axiom all m:Message,
      Equal(Compose(Source(m),Destination(m),
          Text(m)),m);
  record(s,d:Userid; t:CString);
end;

function Source(m:Message):UserId =
begin
  cexit Source(m) = m.s;
  result := m.s;
end;

function Destination(m:Message):UserId =
begin
  cexit Destination(m) = m.d;
  result := m.d;
end;

function Text(m:Message):CString =
begin
  cexit Text(m) = m.t;
  result := m.t;
end;
```

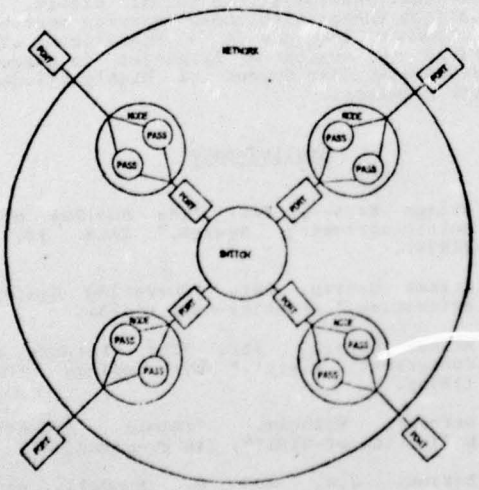


Figure 3

```
function Compose(s,d:UserId; t:CString)
    :Message =
begin
  cexit Compose(s,d,t) = Message(s,d,t);
  result := Message(s,d,t);
end;

function Equal(m1,m2:Message):boolean =
begin
  exit Equal(m1,m2) iff Equal(m2,m1);
  cexit Equal(m1,m2) iff
      m1.s=m2.s and m1.d=m2.d and m1.t=m2.t;
  result := m1.s=m2.s and m1.d=m2.d
      and m1.t=m2.t;
end;
```

In the functions that are permitted access to the internal structure of messages, centry and cexit specifications also are permitted access to the internal structure, but entry and exit specifications are not. Entry and exit specifications are visible externally, to the routines that call the functions, but centry and cexit specifications are not. This prevents the external specifications from revealing the internal structure of messages. In a function the local variable with the reserved name "result" is the value assigned to the function upon exit. It can be used in the same way as any other local variable. In the function compose, message(s,d,t) is another example of the type name used to construct an object of that type. Message(s,d,t) creates a message with successive fields equal to s, d, and t.

This completes the program and its specifications except for assigning values to the pending constants nusers, csize, and timeslice. The program at this stage of development can be written as

```
program network(var upa:PortArray) =
begin
  block all i,j:userid,
      ProperDelivery(i,j,upa);
  var npa:PortArray;
  cobegin
    Node(upa(i),npa(i),i) each i : UserId;
    switch(npa);
  end;
end;

type PortArray = array(UserId) of Port;

type UserId = integer[1..NUsers];
const NUsers:integer = pending;

type Port = record(Get,Put:Line);
type Line = buffer(CSize) of Message;
const CSize:integer = pending;

define ProperDelivery(i,j,pa) =
  mail(pa(j).put.outto,i,j)
      sub mail(pa(i).get.infrom,i,j);

function mail(ms:MessageSequence;i,j:UserId)
    :MessageSequence =
begin
  exit (assume mail(ms,i,j) =
    if ms=MessageSequence()
    then MessageSequence()
    else if i = Source(first(ms)) and
          j = Destination(first(ms))
      then MessageSequence(first(ms))
          @ Mail(nonfirst(ms),i,j)
      else Mail(nonfirst(ms),i,j)
    fi   fi);
end;

type MessageSequence = sequence of Message;
```

```
process Node(var up,np:Port;i:UserId) =
begin
  block up.Put.outto sub np.Put.infrom
    and np.Get.outto sub up.Get.infrom;
  cobegin
    Pass(np.Put,up.Put,i,Depart);
    Pass(up.Get,np.Get,i,Arrive);
  end;
end;

process Pass(var x,y:Line; i:UserId;
    d:Direction) =
begin
  block y.outto sub x.infrom;
  var m:Message;
  loop
    assert y.outto sub x.infrom;
    receive m from x;
    trace i = if d = Depart
        then Destination(m)
        else Source(m) fi;
    send m to y;
  end;
end;

type direction = (Arrive, Depart);

process Switch(var npa:PortArray) =
begin
  block all i,j:UserId,
      ProperDelivery(i,j,npa);
  var m:Message;
  var k:UserId;
  cond DestinationErr;
  keep Destination(m) in [1..NUsers]
      otherwise DestinationErr;
  loop
    k := 1;
    loop
      if k > NUsers then leave end;
      assert all i,j:UserId,
          ProperDelivery(i,j,npa);
      await
      on receive m from npa(k).Get:
          send m to npa(destination(m)).Put;
      after Timeslice: ;
      when
      is DestinationErr: ;
      end;
      k := k + 1;
    end;
  end;
end;

const TimeSlice:integer = pending;

type Message<Source,Destination,Text,Compose,
    Equal> =
begin
  axiom all m:Message,
    Equal(Compose(Source(m),Destination(m),
        Text(m)),m);
  record(s,d:Userid; t:CString);
end;

type CString = sequence (100) of char;

function Source(m:Message):UserId =
begin
  cexit Source(m) = m.s;
  result := m.s;
end;

function Destination(m:Message):UserId =
begin
  cexit Destination(m) = m.d;
  result := m.d;
end;

function Text(m:Message):CString =
begin
  cexit Text(m) = m.t;
  result := m.t;
end;

function Compose(s,d:UserId; t:CString)
    :Message =
begin
  cexit Compose(s,d,t) = Message(s,d,t);
  result := Message(s,d,t);
end;

function Equal(m1,m2:Message):boolean =
begin
  exit Equal(m1,m2) iff Equal(m2,m1);
  cexit Equal(m1,m2) iff
      m1.s=m2.s and m1.d=m2.d and m1.t=m2.t;
  result := m1.s=m2.s and m1.d=m2.d
      and m1.t=m2.t;
end;
```

There are many details of Gypsy that this example does not illustrate, but the development of this program and its specifications provides a good overview of the philosophy and capabilities of the language.

Conclusion

Gypsy has a number of important and distinctive aspects. It is a high-level language for general purpose computing that also supports the development of systems programs. It includes facilities for concurrency and timing, execution in imperfect run-time environments, and an access control mechanism. Gypsy includes extensive and powerful facilities for expressing functional specifications of its programs and of the units from which the program is structured. All constructs in Gypsy are verifiable either by formal proof or run-time validation. Run-time validation can be used effectively to reduce the size and complexity of the formal proofs. Facilities are provided for decomposing both routines and data into small, logically meaningful, units that can be verified independently. This modularity greatly enhances the practical feasibility of formal proofs. We believe that integrating these features smoothly into a common language is a significant step forward in the design of languages to support the systematic development of highly reliable computer programs.

Bibliography

[1] Brinch Hansen, Per. "The Nucleus of a Multiprogramming System," CACM 13, 4 (1970).

[2] Brinch Hansen, Per. "Operating Systems Principles," Prentice-Hall (1973).

[3] Brinch Hansen, Per. "The Purpose of Concurrent Pascal," Proceedings ICRS (1975).

[4] Burger, Wilhelm. "Formal Semantic Definition of GYPSY", (in preparation).

[5] Buxton, J.N. and B. Randell, eds. Software Engineering Techniques, NATO Science Committee (1970).

[6] Clint, M. "Program Proving: Co-routines," Acta Informatica, 2 (1973).

[7] Dahl, O. -J. "Notes on Data Structuring," Dahl, Dijkstra, and Hoare, Structured Programming, Academic Press (1972).

[8] Dijkstra, Edsger W., "Guarded Commands, Nondeterminacy, and Formal Derivation of Programs," CACM 18, 8 (1975).

[9] Flon, Lawrence. "A Survey of Some Issues Concerning Abstract Data Types," Technical Report, Carnegie-Mellon (1974).

[10] Good, D.I., and L.C. Ragland. "Nucleus--A Language for Provable Programs," Program Test Methods, Hetzel (ed.), Prentice-Hall (1973).

[11] Igarashi, S., R.L. London, and D.C. Luckham. "Automatic Program Verification I: A Logical Basis and Its Implementation," Report ISI/RR-73-11, Inf. Sci. Inst. USC (1973).

[12] Jensen, Kathleen and Niklaus Wirth. "Pascal User Manual and Report," Springer Verlag (1974).

[13] Liskov, Barbara and Stephen Zilles. "An Approach to Abstraction," Computation Structures Group Memo 88, MIT (1973).

[14] Lyle, Don M. "A Hierarchy of High Order Languages for Systems Programming," Proc. of SIGPLAN Symp. on Languages for Systems Implementation (1971).

[15] Moriconi, Mark S. "An Interactive System for Incremental Program Design and Verification," (in preparation).

[16] Randall, B., "System Structure for Software Fault Tolerance," Proceedings ICRS (1975).

[17] Stucki, L.G. "Testing Impact on the Future of Software Engineering," Proc. of Fourth Texas Conf. on Computing Systems, University of Texas (1975).

[18] van Wijngaarden, A., et al. "Report on the Algorithmic Language ALGOL 68," Numerische Mathematik, 14 (1969).

[19] Wulf, W.A., D.B. Russell, and A.N. Habermann. "BLISS: A Language for Systems Programming," CACM 14, 12 (1971).

[20] Wulf, W., R. Levin, and C. Pierson. "Overview of the Hydra Operating System Development," Proc. of Fifth Symp. on Operating Systems Principles, (1975).

[21] Wulf, W.A., R.L. London, and Mary Shaw. "Abstraction and Verification in Alphard", preliminary draft, Carnegie-Mellon (1976).

[22] Zahn, Charles T., "A Control Statement for Natural Top-Down Structured Programming," Symp. on Programming Languages (1974).

LIST OF ATTENDEES

Invitational Software V & V Conference

Aug 3,4,5 1976

Lt James Boyd
RADC/OCSP
Griffiss AFB NY 13441
315-330-4481

Mr. J. Brookshire
U.S. Army Missile Command
DRMSI-RGG
Redstone Arsenal AL 35809
205-876-2601

Bill Carlson
Defense Advanced Research Projects Agency
1400 Wilson Blvd
Arlington VA 22209
Autovon 224-5037

Mr. M.E. Champaign
CCCTC-WAD Code C420
11440 Isaac Newton Square N.
Reston VA 22090
703-437-2326

Dr. T. E. Cheatham
Aiken Comp. Lab
Harvard University
Cambridge MA 02138
617-495-3989

Prof. Roger Cheung
Dept. of Comp. Science
Northwestern University
Evanston IL 60201
312-492-5246

Lori Clarke
Dept of Comp Science
University of Massachusettes
Amherst MA 01002

Roy Coppinger
H.O.S.
843 Mass. Ave
Cambridge MA 02139
617-661-8900

Dr. Edward Cragg
CTEC
7777 Leesburg Pike
Falls Church VA 22041
703-821-3700

Steve Crocker
U.S.C. Information Sci. Institute
4676 Admiralty Way
Marina Del Rey CA 90291
213-822-1511

Dr. John Darringer
IBM Research Center Lab
P.O. Box 218
Yorktown Heights NY 10598
914-628-1880

Donald DeVorkin
C.S. Draper Lab
75 Cambridge Parkway
Cambridge MA 02142
617-258-1225

Dr. Bernard Elspas
Comp. Sci. Group
333 Ravenswood Ave
Menlo Park CA 94025
415-326-6200

Carl Engelman
MITRE Corp.
P.O. Box 208
Bedford MA 01730
617-271-2805

Carolyn Gannon
General Research Corp.
5383 Hollister Ave
P.O. Box 3587
Santa Barbara CA 93105
805-964-7724

John B. Glore
MITRE Corp.
P.O. Box 208
Bedford MA 01730
617-271-3022

Dr. Donald Good
Comp. Science Dept.
University of Texas
Austin TX 78712
471-4353

Robert M. Graham
University of Mass.
Coins Graduate Research Center
Amherst MA 01002

Walter Graves
TRW
One Space Park
Bldg 90, Rm 2876
Redondo Beach CA 90278
213-535-3531

Paul Hilfinger
Dept of Comp Science
Carnegie-Mellon University
Pittsburg PA 15213
412-687-5861

R.H. Hoffman
TRW
16011 El Camino Real
Houston TX 77062
713-333-3133

Michael Ikezawa
LOGICON, Inc.
255 West Fifth St
San Pedro CA 90733
213-831-0611

Capt Jack Ives
RADC/ISIS
Griffiss AFB NY 13441
315-330-7010

Maj Ernest Jackson
Patriot Proj. Office
Attn: AMCPN-T-ES
Redstone Arsenal AL 35809
Autovon 742-3755

Maj Ben A. Johnson
BMDATC Attn: ATC-P
Box 1500
Huntsville AL 35807
205-895-4114

Larry Johnson
LOGICON, Inc.
P.O. Box 2566
Framingham MA 01701
617-274-6333

David Kallman
MITRE Corp.
Box 208
Bedford MA 01730
617-271-2040

Col Robert Krutz
RADC/IS
Griffiss AFB NY 13441
315-330-2204

Frank LaMonica
RADC/ISIM
Griffiss AFB NY 13441
315-330-7834

Michael Landes
RADC/ISM
Griffiss AFB NY 13441
315-330-2672

Larry Lombardo
RADC/ISIM
Griffiss AFB NY 13441
315-330-2672

Prof. D.C. Luckham
Stanford University
Palo Alto CA 94305
415-497-4971

Ann Marmour-Squires
National Security Agency
R142
Ft Meade MD 20755
301-688-7125

James Meehan
LOGICON, Inc.
255 West Fifth St
San Pedro CA 01701
213-831-0611

John K. Miller
MITRE Corp.
P.O. Box 208 M.S. B070
Bedford MA 01730
217-271-2036

Charles McClure
MITRE Corp.
P.O. Box 208
Bedford MA 01730
617-271-2175

Dr. Edward Miller
Science Applications Inc.
244 Kearny St
San Francisco CA 94108

Lt Col McKemie
AFOSR/NM
Bldg 410
Bolling AFB Wash DC 20332
202-693-0028

John McLean
RADC/ISIS
Griffiss AFB NY 13441
315-330-7010

Dr. David Musser
USC Information Sciences
4676 Admiralty Way
Marina Del Rey CA 90291
213-822-1511

Lawrence Nicola
Science Applications Inc.
2109 West Clinton Ave
Huntsville AL 35805
205-533-5900

Don Nordman
GRC
RD #3
Rome NY 13440
315-336-2545

Leon Osterweil
Dept of Comp Sci
University of Colorado
Boulder CO 80309
303-492-8787

R. Pennington
General Research Corp.
5383 Hollister Ave
Santa Barbara CA 93105
805-964-7724

Barry Press
TRW
One Space Park Dr
Redondo Beach CA 90278
213-535-3535

P. Radatz
LOGICON, Inc.
255 West Fifth St
P.O. Box 471
San Pedro CA 90733
213-831-0611

Don Reifer
Aerospace Corp
P.O. Box 92957
Los Angeles CA 90009
213-648-5890

David Reilly
Hughes Aircraft Company
Box 3310
Fullerton CA 92634
714-871-3232

Donald Roberts
RADC/ISIS
Griffiss AFB NY 13441
315-330-7546

Dr. Robinson
Link Hall
Syracuse University
Syracuse NY 13441
315-423-3159

Raymond Rubey
SOFTECH
4130 Linden Ave
Dayton OH 45432
513-252-2834

Sabina Saib
General Research Corp.
Santa Barbara CA 93105
805-964-7724

Steve Schanzer
Analytic Services
5613 Leesburg Pike
Falls Church VA 22041
703-820-2830

Val Schorre
Systems Development Corp.
2500 Colorado Ave
Santa Monica CA 90406
213-829-1146

Frank Sliwa
RADC/ISIM
Griffiss AFB NY 13441
315-330-7011

Dr. Jay Spitzen
S.R.I.
333 Ravenswood Ave
Menlo Park CA 94025
415-326-6200 X3044

Robert Stover
RADC/ISIS
Griffiss AFB NY 13441
315-330-7010

Leon Stucki
McDonnell-Douglas Astronautics
5301 Balsa Ave
Huntington Beach CA 92647
714-896-3774

Capt Alan Sukert
RADC/ISIS
Griffiss AFB NY 13443
315-330-4325

Windsor Thomas
RADC/OSCP
Griffiss AFB NY 13441
315-330-4481

Judy Townley
Aiken Comp Lab
Harvard University
Cambridge MA 02138
617-495-3751

Capt W. White
ESD/MCIT
Hanscom AFB MA 01731
617-861-5391

Russell Wilson
SOFTECH
81 Avalon Road
Newton MA 02168
617-890-6900

Richard Weishert
General Research Corp.
5383 Hollister Ave
P.O. Box 3587
Santa Barbara CA 93105
805-964-7728

Dr. Steven Yau
Dept of Comp Sci
Northwestern University
Evanston IL 60201
312-492-3641

Donald Young
ESD/XRI
Hanscom AFB MA
617-271-2756

Selby Young
Kappa Systems Inc.
1409 Potter Dr.
Colorado Springs CO 80909
303-597-1900